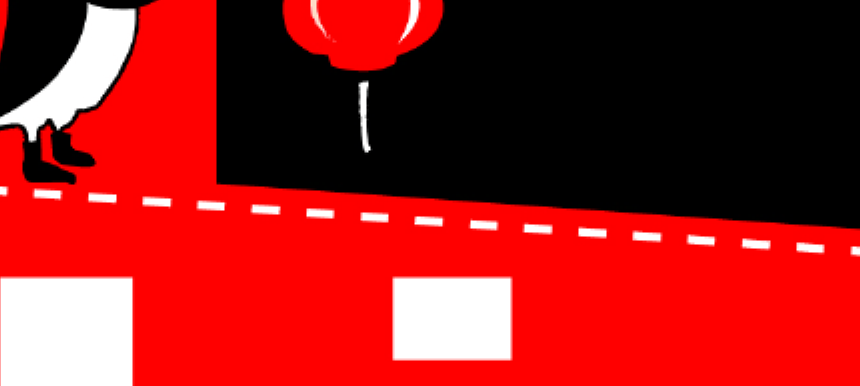
Linux典藏大系




国内第一本全面、系统、深入介绍Linux系统移植技术的图书
注重实战，通过15个典型案例深入剖析Linux系统移植的方法



# Linux系统移植

刘 刚
赵剑川 等编著




17.5小时配套多媒体教学视频
44小时Linux专题视频讲座（免费赠送）

◎ 内容全面：覆盖Linux内核、文件系统、驱动及数据库等各种系统的移植
◎ 讲解详细：所有编译过程都附有编译命令，并对复杂命令给出了详细说明
◎ 循序渐进：遵循原理分析→代码分析→编译→测试→移植的学习顺序
◎ 技巧性强：贯穿大量的经验和技巧，并对容易出错的地方给出了专门的提示
◎ 注重实战：通过典型案例，让读者深入体验Linux系统移植的方法和全过程

清华大学出版社

Linux 典藏大系

# Linux 系统移植

刘刚　赵剑川　等编著

清华大学出版社
北　京

## 内 容 简 介

本书全面、系统、由浅入深地介绍了 Linux 系统移植的各方面知识。书中的每个章节都有相应的实例编译或移植过程，每个移植实例都具有代表性，在实际应用和开发中有很高的价值。

本书附带 1 张光盘，内容为本书重点内容的教学视频和本书涉及的源代码。另外，还赠送了大量的 Linux 学习视频和其他学习资料。

本书分为 4 篇。第 1 篇简单介绍了 Linux 内核和嵌入式 Linux 系统开发环境搭建；第 2 篇介绍了一个最基本的嵌入式系统的组成部分、Bootloader 移植、内核移植和文件系统移植；第 3 篇介绍了 LCD、触摸屏、USB、网卡、音频、SD 卡、NandFlash 等流行的设备驱动移植过程；第 4 篇从嵌入式产品角度出发，介绍了 GUI、Qtopia、嵌入式数据库 Berkeley DB 和 SQLite、嵌入式 Web 服务器 BOA 和 Thttpd、JVM 虚拟机的移植及目前流行的 VoIP 技术和相关协议。

本书适合嵌入式 Linux 系统入门人员、Linux 系统开发和移植、系统分析师等相关人员阅读，也适合作为大、中专院校相关专业的实验教材使用。



图书在版编目（CIP）数据

Linux 系统移植 / 刘刚，赵剑川等编著. —北京：清华大学出版社，2011.1
（Linux 典藏大系）
ISBN 978-7-302-23922-2

Ⅰ. ①L…　Ⅱ. ①刘…　②赵…　Ⅲ. ①Linux 操作系统　Ⅳ. ①TP316.89

中国版本图书馆 CIP 数据核字（2010）第 192689 号

**责任编辑：**夏兆彦
**责任校对：**徐俊伟
**责任印制：**
**出版发行：**清华大学出版社　　**地　　址：**北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编：**100084
**社 总 机：**010-62770175　　**邮　　购：**010-62786544
**投稿与读者服务：**010-62795954，jsjjc@tup.tsinghua.edu.cn
**质量反馈：**010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 刷 者：**
**装 订 者：**
**经　　销：**全国新华书店
**开　　本：**185×260　　**印　　张：**34.25　　**字　　数：**855 千字
（附光盘 1 张）
**版　　次：**2011 年 1 月第 1 版　　**印　　次：**2011 年 1 月第 1 次印刷
**印　　数：**1～ 000
**定　　价：**　　元

产品编号：039981-01

# 前　言

随着各种芯片技术的发展，各种嵌入式产品也如雨后春笋一般地出现了。目前，嵌入式产品应用领域涉及移动通信、汽车、医疗、家电等很多领域。而且，如今的嵌入式硬件的速度和容量越来越接近于PC，因此在这些嵌入式产品上运行操作系统就成为了可能。一直以来，很多企业花费了巨大成本研发了大量运行在PC上的软件产品。如果将这些优秀的软件应用在嵌入式系统中，将会成为快速开发嵌入式系统，降低嵌入式产品开发成本，提高软件稳定性和安全性的重要途径。

目前，国内图书市场上还鲜见专门介绍Linux系统移植的图书。为了给广大Linux开发人员和爱好者学习Linux系统移植提供一些有价值的参考资料，笔者花费一年多的时间编写了本书。

本书注重实践，包含了丰富的移植实例，这些实例各具特点，从基础的系统组成到设备驱动，再到高级应用，适合各个层面的读者学习和研究。本书中的实例是笔者根据实际项目中嵌入式产品的功能需求，专门选择的具有代表性的开源软件进行移植，包含了常见的嵌入式产品的最小系统组成部分移植，同时选择了应用比较多的数据库、Web服务器、GUI等进行移植。笔者通过亲自体会每次编译和移植过程，详细说明移植的细节，对移植过程中遇到的问题也给出了解决方法。本书最后还介绍了VoIP技术，并结合源码分析了VoIP的实现，同时还介绍了VoIP的详细编译过程。本书是笔者从事嵌入式开发的经验总结，希望能给目前从事嵌入式研发和学习的读者提供最有效的帮助，能使读者的嵌入式系统最快地运行起来，使读者在最短的时间内成功移植开源软件。

本书使用的源代码均为开源代码，读者可以从对应的官方网站获得。本书对于源码的重要部分进行了详细的分析，建议读者在阅读时对应源码进行阅读效果会更好。

## 本书特色

### 1．多媒体语音视频讲解，高效、直观

笔者对本书重点内容专门录制了多媒体教学视频，这将会大大提高读者的学习效率。

### 2．编译过程详细

本书的编译过程都附有详细的编译命令，对于复杂的命令均给出了说明，方便读者实际操作。读者可以边阅读本书，边动手进行实验。

### 3．内容全面、选材具有特点

本书介绍了最小系统的引导程序移植、内核移植、文件系统移植、各种驱动移植等内

容。另外，本书还专门介绍了嵌入式数据库、嵌入式 GUI、嵌入式 Web 服务器、嵌入式 JVM、VoIP 技术等内容。对于数据库、GUI、Web 服务器分别选择了两种进行介绍，读者可以从性能上进行对比，然后应用在自己的项目中。

### 4．内容由浅入深、循序渐进，可操作性强

本书按照由浅入深、循序渐进的梯度安排内容，适合各个层次的读者阅读。书中每章内容都遵循原理分析→代码分析→编译→测试→移植的学习顺序，具有较强的可操作性。

### 5．贯穿了大量的编译技巧，可迅速提升移植水平

本书在讲解编译过程时贯穿了大量的编译技巧，并针对移植过程中的编译错误介绍了如何发现错误的源头，同时给出了解决方法。这则有利于读者解决类似的编译问题，提升系统移植的水平。

### 6．详细介绍了流行工具的使用

本书介绍了在开发中使用 Eclipse 和 VC++ 6.0，这两种工具分别为 Java 程序员和 C++ 程序员最熟悉的工具。书中介绍了在 Linux 下安装 Eclipse、使用 Eclipse 开发 C++项目及使用 VC++ 6.0 开发的基本方法等。

## 本书内容及知识体系

### 第1篇　系统移植基础篇（第1、2章）

本篇介绍了系统移植的基础。首先对 Linux 内核进行了简单介绍，然后介绍了系统移植环境的搭建。通过对本篇内容的学习，读者可以对 Linux 系统有初步认识，能掌握嵌入式 Linux 开发工具的使用，能正确搭建开发平台，能够制作编译好的嵌入式系统。

### 第2篇　系统移植技术篇（第3～5章）

本篇介绍了一个最基本的嵌入式系统的组成部分、Bootloader、内核和文件系统的移植。学习完本篇内容后，读者能够动手独立编译和移植一个基本的嵌入式系统。

### 第3篇　系统移植驱动篇（第6～12章）

本篇介绍了各种驱动的移植，包括 LCD、触摸屏、USB、网卡、音频、SD 卡、NandFlash 等流行的设备驱动的移植过程。通过对本篇内容的学习，读者对嵌入式 Linux 驱动移植将会有一定的认识，可以基本掌握驱动的移植步骤，能完成简单的驱动移植。

### 第4篇　系统移植高级篇（第13～20章）

本篇从嵌入式产品的角度出发，介绍了系统移植中各种类型的高层软件移植，包括 GUI、数据库、Web 服务器、虚拟机的移植，最后还介绍了目前流行的 VoIP 技术，并结合源码介绍了 VoIP 相关协议和编译方法。通过学习本篇内容，读者可以掌握很多移植技巧，能够将这些实例应用到自己的项目中。

## 本书读者对象

- 嵌入式移植人员；
- 嵌入式专业的学生；
- 嵌入式实验指导老师；
- 嵌入式培训学员和老师；
- 系统分析师；
- 项目研发人员。

## 本书作者及编委会成员

本书由刘刚和赵剑川主笔编写，其他参与编写的人员有毕梦飞、蔡成立、陈涛、陈晓莉、陈燕、崔栋栋、冯国良、高岱明、黄成、黄会、纪奎秀、江莹、靳华、李凌、李胜君、李雅娟、刘大林、刘惠萍、刘水珍、马月桂、闵智和、秦兰、汪文君、文龙。在此一并表示感谢。

本书编委会成员有欧振旭、陈杰、陈冠军、项宇峰、张帆、陈刚、程彩红、毛红娟、聂庆亮、王志娟、武文娟、颜盟盟、姚志娟、尹继平、张昆、张薛。

## 本书技术支持

您在阅读本书的过程中若碰到什么问题，请通过以下方式联系我们，我们会及时地答复您。

E-mail：bookservice2008@163.com（编辑）

论坛网址：http://www.wanjuanchina.net

编著者

# 目　　录

## 第 1 篇　系统移植基础篇

## 第 3 篇　系统移植与驱动篇

# 第 1 篇　系统移植基础篇

# 第 1 章　Linux 内核介绍

在进行系统移植的时候，移植的核心就是内核移植。内核移植不仅影响内核的功能，而且还影响到整个系统的性能。因此，了解 Linux 内核，有利于开发人员进行系统裁剪和移植。下面主要针对 Linux 内的 5 个重要部分（系统调用接口、进程管理、内存管理、虚拟文件系统和设备驱动程序）进行介绍。

## 1.1　系统调用接口

系统调用是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口获得操作系统内核提供的服务。例如，用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件；通过时钟相关的系统调用获得系统时间或设置系统时间；通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

### 1.1.1　Linux 系统调用

所有的操作系统在内核里都有一些内建的函数，这些函数完成对硬件的访问和对文件的打开、读、写、关闭等操作。Linux 系统中称这些函数叫做“系统调用”（即 systemcall）。这些函数实现了将操作从用户空间转换到内核空间，有了这些接口函数，用户就可以方便地访问硬件。例如，在用户空间调用 open()函数，则会在内核空间调用 sys_open()。一个已经安装的系统所支持的系统调用都可以在/usr/include/bits/syscall.h 文件里看到。

为了对系统进行保护，Linux 系统定义了内核模式和用户模式。内核模式可以执行一些特权指令并进入用户模式，而用户模式则不能进入内核模式。Linux 将程序的运行空间也分为内核空间和用户空间，它们分别运行在不同的级别上。在逻辑上，它们是相互隔离的。系统调用规定用户进程进入内核空间的具体位置。在执行系统调用时，程序运行空间将会从用户空间转移到内核空间，处理完毕后再返回到用户空间。

### 1.1.2　用户编程接口

用户编程接口是为用户编程过程提供的各种功能库函数，如分配空间、拷贝字符、打开文件等。Linux 用户编程接口（API）遵循了在 UNIX 中最流行的应用编程界面标准——POSIX 标准。它与系统调用之间存在一定的联系和区别。不同的语言和平台为用户提供了丰富的编程接口，包括网络编程接口、图形编程接口、数据库编程接口等，但这些不是系统调用。

系统调用与用户编程接口它们之间存在联系。一个或者多个系统调用会对应到一个具体的应用程序使用的 API；但是，并非所有的 API 都需要使用到系统调用。

根据《深入理解 Linux 内核》一文中对用户编程接口（API）和系统调用两者区别的描述为，前者只是一个函数定义，说明了如何获得一个给定的服务；而后者是通过软中断向内核态发出一个明确的请求。如果按照层次关系来分，系统调用为底层，而用户编程接口为上层。一个用户编程接口由 0 个或者多个系统调用组成。

### 1.1.3　系统调用与服务例程的对应关系

为了通过系统调用号来调用不同的内核服务例程，系统必须创建并管理好一张系统调用表。该表用于系统调用号与内核服务函数的映射，Linux 用数组 sys_call_table 表示这个表。在这个表的每个表项中存放着对应内核服务例程的指针，而该表项的下标就是该内核服务例程的系统调用号。Linux 规定，在 I386 体系中，处理器的寄存器 eax 用来传递系统调用号。

### 1.1.4　系统调用过程

通常情况下，abc()系统调用对应的服务例程的名字是 sys_abc()。图 1.1 表示了系统调用和应用程序、对应的封装例程、系统调用处理程序及系统调用服务例程之间的关系。下面使用一个例子来简单说明系统调用过程。

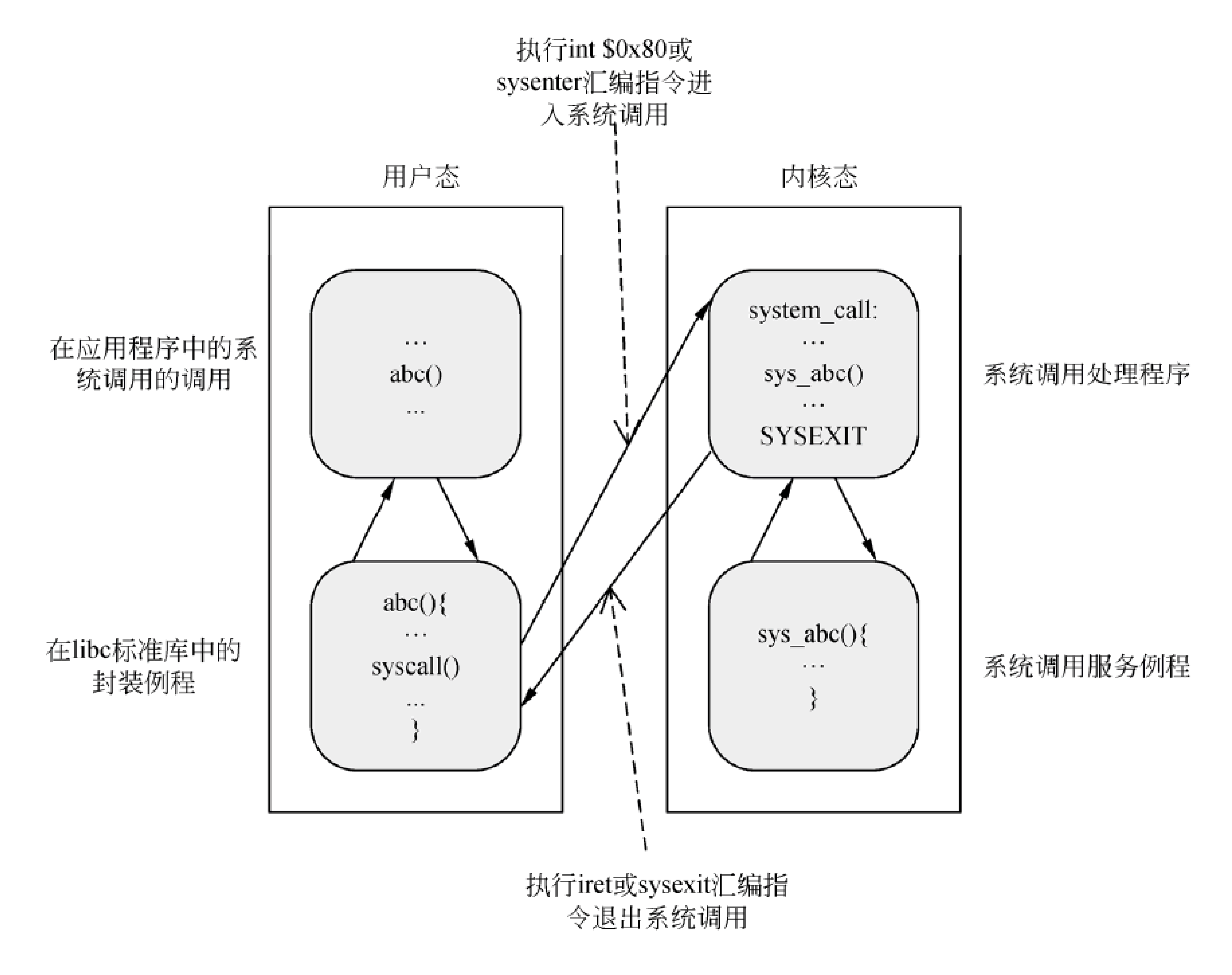



图 1.1　系统调用的处理过程

（1）用户程序中调用库函数 abc()。

（2）系统加载 libc 库调用索引和参数后，执行 int $0x80 或者 sysenter 汇编指令进入系统调用，执行 system_call()函数。

（3）system_call()函数根据传递过来的参数处理所有的系统调用。使用 system_call_table[参数]执行系统调用。

（4）系统调用返回。

（5）执行 iret 或者 sysexit 汇编指令两种方式退出系统调用，并调用 resume_userspace()函数进入用户空间。

（6）继续在 libc 库中执行，执行完成后返回到用户应用程序中。

执行 int $0x80 或者 sysenter 汇编指令两种方式进入系统调用，在老版本的 Linux 内核中只支持 int $0x80 中断方式。执行 iret 或者 sysexit 汇编指令两种方式退出系统调用，如图 1.1 虚线指引线所示。

### 1.1.5　系统调用传递的参数

系统调用中输入输出的参数为实际传递的值或者是用户态进程的地址，或者是指向用户态函数指针的数据结构地址。传递的参数放在寄存器 eax 中，即系统调用号。寄存器传递参数的个数满足两个条件：

- 参数的长度不超过寄存器的长度，如果是 32 位平台不超过 32 位，64 位平台不超过 64 位。
- 不包括 eax 中的系统调用号，参数的个数不超过 6 个。

内核在为用户请求提供服务时，会检查所有的系统调用参数。检查参数时，主要对参数的权限和参数表示的地址空间的有效性进行验证。

## 1.2　进 程 管 理

进程管理包括创建进程、管理进程及删除进程。进程管理是 Linux 内核的重要部分，对系统的核心资源进行管理。做好系统移植需要对这部分知识有一定的了解。

### 1.2.1　进程

进程是程序执行时的一个实体。程序包含指令和数据，而进程包含程序计数器和全部 CPU 寄存器的值。进程的堆栈中存储着一些数据，如子程序参数、返回地址及变量之类的临时数据。当前的执行程序（进程）包含着当前处理器中的活动状态。

Linux 是一个多处理操作系统，进程拥有独立的权限和单一职责。如果系统中某个进程发生崩溃，它不会影响到另外的进程。每个进程都运行在其各自独立的虚拟地址空间中，只有通过核心控制下可靠的进程通信机制，它们之间才能发生通信。进程通信机制包括：管道、信号、信号量、消息队列等。

从内核的观点看，进程的目的就是担当分配系统资源的实体。系统资源包括 CPU 时间、

内存等。因此，进程管理的最终目的就是在进程畅通执行的条件下，合理分配系统资源给不同的进程。

当一个进程创建时，它几乎与父进程相同，即是父进程地址空间的一个（逻辑）备份。从进程创建系统调用的下一条指令开始，执行与父进程相同的代码。虽然父子进程含有共享的程序代码，但是分别拥有独立的数据备份。因此子进程对堆和栈中的数据进行修改时，对父进程的数据是不会有影响的。

## 1.2.2　进程描述符

内核对进程的优先级、进程的状态、地址空间等采用进程描述符表示。在 Linux 内核中，进程用相当大的一个称为 task_struct 的结构表示。下面是从 linux-2.6.29\include\linux\sched.h 中摘抄出来的进程描述的部分信息。

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
    int lock_depth;     /* BKL lock depth */
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    unsigned char fpu_counter;
    s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
    unsigned int policy;
    cpumask_t cpus_allowed;
    struct list_head tasks;
    struct mm_struct *mm, *active_mm;

/* task state */
    struct linux_binfmt *binfmt;
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */

    unsigned int personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;

    struct task_struct *real_parent; /* real parent process */
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    struct list_head children;  /* list of my children */
    struct list_head sibling;  /* linkage in my parent's children list */
    struct task_struct *group_leader;  /* threadgroup leader */
    struct list_head ptraced;
    struct list_head ptrace_entry;
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;

    struct completion *vfork_done;     /* for vfork() */
```

```
    int __user *set_child_tid;      /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */
    ...
};
```

简单对上述内容进行描述。

- state：表示进程的状态，–1 代表“不能运行”，0 代表“运行”，>0 代表“停止”。
- flags：定义了很多指示符，表明进程是否正在被创建（PF_STARTING）或退出（PF_EXITING），或是进程当前是否在分配内存（PF_MEMALLOC）。可执行程序的名称（不包含路径）占用 comm（命令）字段。
- 每个进程都会被赋予优先级（称为 static_prio），但进程的实际优先级是基于加载及其他几个因素动态决定的。优先级值越低，实际的优先级越高。
- tasks 字段提供了链接列表的能力。它包含一个 prev 指针（指向前一个任务）和一个 next 指针（指向下一个任务）。

进程的地址空间由 mm 和 active_mm 字段表示。mm 代表的是进程的内存描述符，而 active_mm 则是前一个进程的内存描述符（为改进上下文切换时间的一种优化）。

### 1.2.3　进程状态

进程描述符中 state 字段描述进程当前的状态。它由一组标志组成，其中每个标志描述一种可能的进程状态。在 2.6 内核中，进程只能处于这些状态中的一种。下面分别对这些状态进行描述。

- 可运行状态（TASK_RUNNING）：进程处于运行（它是系统的当前进程）或者准备运行状态（它在等待系统将 CPU 分配给它）。
- 等待状态（WAITING）：进程在等待一个事件或者资源。Linux 将等待进程分成两类：可中断的等待状态（TASK_TNTERRUPTIBLE）与不可中断的等待状态（TASK_UNINTERRUPTIBLE）。可中断等待进程可以被信号中断；不可中断等待进程直接在硬件条件等待，并且任何情况下都不可中断。
- 暂停状态（TASK_STOPPED）：进程被暂停，通常是通过接收一个信号（SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU）。正在被调试的进程可能处于停止状态。
- 僵死状态（EXIT_ZOMBIE）：进程的执行被终止，但是，父进程还没有发布 wait4()或 waitpid()系统调用返回有关死亡进程的信息。

### 1.2.4　进程调度

Linux 进程调度的目的就是调度程序运行时，要在所有可运行状态的进程中选择最值得运行的进程投入运行。每个进程的 task_struct 结构中有 policy、priority、counter 和 rt_priority 这 4 项是选择进程的依据。

其中，policy 为进程调度策略，用于区分普通进程和实时进程，实时进程优先于普通进程运行；priority 是进程（包括实时和普通）的静态优先级；counter 是进程剩余的时间片，它的起始值就是 priority 的值；因为 counter 用于计算一个处于可运行状态的进程值的运行的程度 goodness，所以 counter 也被看作是进程的动态优先级。rt_priority 是实时进程

特有的优先级别，用于实时进程间的选择。

### 1．Linux进程分类

Linux 在执行进程调度的时候，对不同类型的进程采取的策略也不同，一般将 Linux 分为以下 3 类。

- 交互式进程：这类进程经常和用户发生交互，所以花费一些时间等待用户的操作。当有用户输入时，进程必须很快地激活。通常要求延迟在 50～150 毫秒。典型的交互式进程有：控制台命令、文本编辑器、图形应用程序等。
- 批处理进程（Batch Process）：这类进程不需要用户交互，一般在后台运行。所以不需要非常快的反应，它们经常被调度期限制。典型的批处理进程有编译器、数据库搜索引擎和科学计算等。
- 实时进程：这类进程对调度有非常严格的要求，这种类型的进程不能被低优先级进程阻塞，并且在很短的时间内做出反应。典型的实时进程有音视频应用程序、机器人控制等。

### 2．Linux进程优先级

Linux 系统中每一个普通进程都有一个静态优先级。这个值会被调度器用来作为参考来调度进程。在内核中调度的优先级区间为[100,139]，数字越小，优先级越高。一个新的进程总是从它的父进程继承此值。Linux 进程优先级还包括动态优先级、实时优先级等，各个进程优先级描述如下。

- 静态优先级（priority）：被称为“静态”是因为它不随时间而改变，只能由用户进行修改。它指明了在被迫和其他进程竞争 CPU 之前该进程所应该被允许的时间片的最大值（20）。
- 动态优先级（counter）：counter 即系统为每个进程运行而分配的时间片。Linux 用它来表示进程的动态优先级。当进程拥有 CPU 时，counter 就随着时间不断减小。当它递减为 0 时，标记该进程将重新调度。它指明了在当前时间片中所剩余的时间量（最初为 20）。
- 实时优先级（rt_priority）：它的变化范围是从 0～99。任何实时进程的优先级都高于普通的进程。
- Base time quantum：是由静态优先级决定，当进程耗尽当前 Base time quantum，kernel 会重新分配一个 Base time quantum 给它。静态优先级和 Base time quantum 的关系为：

（1）当静态优先级小于 120：

```
Base time quantum (in millisecond) = (140 - static priority) * 20
```

（2）当静态优先级大于等于 120：

```
Base time quantum (in millisecond) = (140 - static priority) * 5
```

### 3．Linux进程的调度算法

在 Linux 系统中，进程作为程序的实体始终运行在系统中，并且进程占用系统资源，

所以进程调度算法优劣将会严重影响到系统的性能。为提高系统性能设计进程调度算法的原则，应该遵循：进程响应尽量快，后台进程的吞吐量尽量大，尽量避免进程“饿死”现象，低优先级和高优先级进程需要尽可能调和。以下为常见的进程调度算法。

- 时间片轮转调度算法（round-robin）：SCHED_RR 用于实时进程。系统使每个进程依次地按时间片轮流执行的方式。
- 优先权调度算法：SCHED_NORMAL 用于非实时进程。每次系统都会选择队列中优先级最高的进程运行。Linux 采用抢占式的优级算法，即系统中当前运行的进程永远是可运行进程中优先权最高的进程。
- FIFO（先进先出）调度算法：SCHED_FIFO 用于实时进程。采用 FIFO 调度算法选择的实时进程必须是运行时间较短的进程，因为这种进程一旦获得 CPU 就只有等到它运行完或因等待资源主动放弃 CPU 时，其他进程才能获得运行机会。

## 1.2.5　进程地址空间

Linux 的虚拟地址空间为 0～4GB。虚拟的 4GB 空间被 Linux 内核分为内核空间和用户空间两部分。将最高的 1GB（从虚拟地址 0xC0000000 到 0xFFFFFFFF）留给内核使用，称为“内核空间”。将较低的 3GB（从虚拟地址 0x00000000 到 0xBFFFFFFF）留给用户进程使用，称为“用户空间”。因为每个进程可以通过系统调用进入内核，因此，Linux 内核空间被系统的所有进程共享，实际上对于某个进程来说，它仍然可以拥有 4GB 的虚拟空间。

其中，很重要的一点是虚拟地址空间，并不是实际的地址空间。在为进程分配地址空间时，根据进程需要的空间进行分配，4GB 仅仅是最大限额而已，并非一次性将 4GB 分配给进程。一般一个进程的地址空间总是小于 4GB 的，可以通过查看/proc/pid/maps 文件来获悉某个具体进程的地址空间。但进程的地址空间并不对应实际的物理页，Linux 采用 Lazy 的机制来分配实际的物理页（“Demand paging”和“写时复制（Copy On Write）的技术”），从而提高实际内存的使用率，即每个虚拟内存页并不一定对应于物理页。虚拟页和物理页的对应是通过映射机制来实现的，即通过页表进行映射到实际的物理页。因为每个进程都有自己的页表，因此可以保证不同进程的相同虚拟地址可以映射到不同的物理页，从而为不同的进程都可以同时拥有 4GB 的虚拟地址空间提供了可能。

内核是系统中优先级最高的部分。内核函数申请动态内存时系统不会推迟这个请求；而进程申请内存空间时，进程的可执行文件被装入后，进程不会立即对所有的代码进行访问。因此内核总是尽量推迟给用户进程分配动态空间。

内核分配空间时，可以通过__get_free_pages()或 alloc_pages 从分区页框分配器中获得页框；通过 kmem_cache_alloc()或 kmalloc()函数使用 slab 分配器为专用或通用对象分配块；通过 vmalloc()或 vmalloc32()函数获得一块非连续的内存区。

与进程地址空间有关的全部信息都包含在一个叫做内存描述符（memeory desvriptor）的数据结构中，其结构类型为 mm_structs 进程描述符的 mm 字段，就是指向这个结构的。

### 1．创建进程地址空间

copy_mm()函数通过建立新进程的所有页表和内存描述符，来创建进程的地址空间。

```
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
    struct mm_struct * mm, *oldmm;
    int retval;

    tsk->min_flt = tsk->maj_flt = 0;
    tsk->nvcsw = tsk->nivcsw = 0;

    tsk->mm = NULL;
    tsk->active_mm = NULL;

/*如果是内核线程的子线程(mm=NULL),则直接退出,此时内核线程mm和active_mm均为NULL*/
    oldmm = current->mm;
    if (!oldmm)
        return 0;

/*内核线程，只是增加当前进程的虚拟空间的引用计数*/
    if (clone_flags & CLONE_VM) {
        /*如果共享内存，将 mm 由父进程赋值给了子进程，2 个进程将会指向同一块内存*/
        atomic_inc(&oldmm->mm_users);
        mm = oldmm;
        goto good_mm;
    }

    retval = -ENOMEM;
    mm = dup_mm(tsk);        /*完成了对 vm_area_struct 和页面表的复制*/
    if (!mm)
        goto fail_nomem;

good_mm:
    /* Initializing for Swap token stuff */
    mm->token_priority = 0;
    mm->last_interval = 0;

/*内核线程的 mm 和 active_mm 指向当前进程的 mm_struct 结构*/
    tsk->mm = mm;
    tsk->active_mm = mm;
    return 0;

fail_nomem:
    return retval;
}
```

### 2. 删除进程地址空间

内核调用 exit_mm()函数释放进程的地址空间。

```
static void exit_mm(struct task_struct * tsk)
{
m_release(tsk, mm);
/*得到读写信号量*/
    down_read(&mm->mmap_sem);
    core_state = mm->core_state;
    if (core_state) {
        struct core_thread self;
/*释放读写信号量*/
        up_read(&mm->mmap_sem);
```

```
        self.task = tsk;
        self.next = xchg(&core_state->dumper.next, &self);

        if (atomic_dec_and_test(&core_state->nr_threads))
            complete(&core_state->startup);

        for (;;) {
            set_task_state(tsk, TASK_UNINTERRUPTIBLE);
            if (!self.task)     /*take 字段可以查看函数 coredump_finish()*/
                break;
            schedule();
        }
        __set_task_state(tsk, TASK_RUNNING);
        down_read(&mm->mmap_sem);
    }
    atomic_inc(&mm->mm_count);
    BUG_ON(mm != tsk->active_mm);
    /* more a memory barrier than a real lock */
    task_lock(tsk);
    tsk->mm = NULL;
    up_read(&mm->mmap_sem);
    enter_lazy_tlb(mm, current);
    /*释放用户虚拟空间的数据结构*/
    clear_freeze_flag(tsk);
    task_unlock(tsk);
    mm_update_next_owner(mm);
    /*递减 mm 的引用计数并是否为 0，如是，则释放 mm 所代表的映射*/
    mmput(mm);
}
```

# 1.3　内 存 管 理

RAM 的一部分被静态地划分给了内核，用来存放内核代码和静态数据结构。RAM 的其余部分称为动态内存（dynamic memory），这不仅是运行用户进程所需的宝贵资源，也是内核所需的宝贵资源。事实上，整个系统的性能取决于如何有效地管理动态内存。

## 1.3.1　内存管理技术

页表（page tables）：进程在读取指令和存取数据时都要访问内存。在一个虚拟内存系统中，所有的地址都是虚拟地址而非物理地址。操作系统维护虚拟地址和物理地址转换的信息，处理器通过这组信息将虚拟地址转换为物理地址。虚拟内存和物理内存被分为适当大小的块，叫做页。为了将虚拟地址转换为物理地址，首先，处理器要找到虚拟地址的页编号和页内偏移量；然后，处理器根据虚拟地址和物理地址的映射关系将虚拟页编号转换为物理页；最后，根据偏移量访问物理页的确定偏移位置。每个物理页面都有一个 struct page 结构，位于 include/linux/mm.h，该结构体包含了管理物理页面时的所有信息，下面给出该结构体的具体描述：

```
typedef struct page {
         struct list_head list;         //指向链表中的下一页
```

```
        struct address_space *mapping;
                        //用来指定我们正在映射的索引节点（inode）
        unsigned long index;            //在映射表中的偏移
        struct page *next_hash;         //指向页高速缓存哈希表中下一个共享的页
        atomic_t count;                 //引用这个页的个数
        unsigned long flags;            //页面各种不同的属性
        struct list_head lru;           //用在 active_list 中
        wait_queue_head_t wait;         //等待这一页的页队列
        struct page **pprev_hash;
                        //指向页高速缓存哈希表中前一个共享的页与 next_hash 相对应
        struct buffer_head * buffers; //把缓冲区映射到一个磁盘块
        void *virtual;
        struct zone_struct *zone;       //页所在的内存管理区
    } mem_map_t;
```

### 1．请求页面调度（Demand Paging）

为了节省物理内存，只加载执行程序正在使用的虚拟页，这种进行访问时才加载虚拟页的技术叫做 Demand Paging。当一个进程试图访问当前不在内存中的虚拟地址时，处理器无法找到引用的虚拟页对应的页表条目。当处理器无法将虚拟地址转换为物理地址时，处理器通知操作系统发生 page fault。出错的虚拟地址无效，则意味着进程试图访问它不应该访问的虚拟地址。这种情况下，操作系统会中断它，从而保护系统中的其他进程。

如果出错的虚拟地址有效，而只是它所在的页当前不在内存中，操作系统应该从磁盘映像中将对应的页加载到内存中。相对内存存取来讲，磁盘存取需要更长的时间，所以进程一直处于等待状态直到该页被加载到内存中。如果系统当前有其他进程可以运行，操作系统将选择其中一个运行；接着将取到的页写进一个空闲页面，并将一个有效的虚拟页条目加到进程的页表中；然后，这个等待的进程重新执行发生内存出错地方的机器指令。本次虚拟内存存取进行时，处理器能够将虚拟地址转换为物理地址，使得进程能够继续运行。Linux 使用 demand paging 技术将可执行映像加载到进程的虚拟内存中，在执行命令时，包含命令的文件被打开，将该文件的内容映射到进程的虚拟内存中。这个过程通过修改描述进程内存映射的数据结构来实现，也叫做内存映射（memory mapping），但实际上只有映像的第一部分真正放在了物理内存中，映像的剩余部分仍然在磁盘上。当映像执行时，它产生 page fault，Linux 使用进程的内存映像表来确定映像的哪一部分需要加载到内存中执行。

### 2．页面置换技术（Swapping）

如果进程需要将虚拟页放到物理内存中，而此时已经没有空闲的物理页，操作系统必须废弃物理空间中的另一页，为该页让出空间。如果物理内存中需要废弃的页来自磁盘上的映像或者数据文件，并且该页没有被写过不需要存储，则该页被废弃。如果进程又需要该页，它可以从映像或数据文件中再次加载到内存中。但如果该页已经被改变，操作系统必须保留它的内容以便以后进行访问。这种也叫做 dirty page ，当它从物理内存中废弃时，被存到一种叫做交换文件的特殊文件中。由于访问交换文件与访问处理器、物理内存的速度相比较慢，操作系统必须判断是将数据页写到磁盘上还是将它们保留在内存中以便下次访问。

如果判断哪些页将被废弃或者交换的算法效率不高，则会发生颠簸（thrashing），这时页不停地被写到磁盘上，然后又被读回，操作系统频繁地处理此读写任务而无法执行实际的工作。Linux 使用 LRU（Least Recently Used，最近最少使用置换算法）的页面技术公平地选择需要从系统中废弃的页面。

伙伴系统算法，该算法用以解决外碎片问题。把所有的空闲页框分为 11 个块链表，每个块链表分别包含 1、2、4、8、16、32、64、128、256、512 和 1024 个连续的页框。对于 1024 个页框的最大请求对应着 4MB 大小的连续 RAM 块。每个块的第一个页框的物理地址是该块大小的整数倍。

非连续内存管理，当对内存区的请求不是很频繁的时候，通过连续的线性地址访问非连续的页框，该方法可以避免外碎片，但是其带来的负面因素是打乱了内核表。非连续内存区的大小必须是 4096 字节的倍数。非连续内存区应用的场合分别有分配数据结构给活动的交换区、分配空间给模块和分配缓冲区给某些 I/O 驱动程序。

## 1.3.2　内存区管理

内存区（memory area）是具有连续的物理地址和任意长度的内存单元序列。伙伴系统采用的是页框作为基本内存区，适合于大内存的请求，对小内存的请求容易造成内碎片。为了解决内碎片的问题，将内存区大小按几何分布划分，也就是将内存区划分成 2 的幂的大小。不论请求的大小为多大时，总能保证内碎片小于内存区的 50%。为此，内核建立了 13 个按几何分布的空闲内存区链表，大小从 32～131 072 字节。伙伴系统的调用为了获得存放新内存区所需的额外页框，同时也为了释放不再包含内存区的页框，用一个动态链表来记录每个页框所包含的空闲内存区。

物理内存被划分为 3 个区来管理，它们是 ZONE_DMA、ZONE_NORMAL 和 ZONE_HIGHMEM。每个区都用 struct zone_struct 结构表示，定义于 include/linux/mmzone.h：

```
typedef struct zone struct {
      /* Commonly accessed fields:*/
         spinlock_t lock;
         unsigned long    free pages;
         unsigned long    pages_min, pages_low, pages_high;
         int              need_balance;
      /* free areas of different sizes*/
          free_area_t free_area[MAX_ORDER];
      /* Discontig memory support fields.*/
         struct pglist_data *zone_pgdat;
         struct page *zone_mem_map;
         unsigned long          zone_start_paddr;
         unsigned long          zone start mapnr;
      /*
      * rarely used fields:
      */
         char                  *name;
         unsigned long          size;
      } zone_t;
```

各个字段的含义如下。

- lock ：用来保证对该结构中其他域的串行访问。

- free_pages ：在这个区中现有空闲页的个数。
- pages_min、pages_low 及 pages_high 是对这个区最少、次少及最多页面个数的描述。
- need_balance：与 kswapd 合在一起使用。
- free_area：在伙伴分配系统中的位图数组和页面链表。
- zone_pgdat：本管理区所在的存储结点。
- zone_mem_map：该管理区的内存映射表。
- zone_start_paddr：该管理区的起始物理地址。
- zone_start_mapnr：在 mem_map 中的索引（或下标）。
- name：该管理区的名字。

### 1.3.3　内核中获取内存的几种方式

操作系统的内存管理方案优劣是决定其效率高低的重要因素，时间与空间常常作为内存管理方案优劣的衡量指标。首先，分配/释放内存是一个发生频率很高的操作，所以它要求有一定的实时性，另外，内存又是一种非常宝贵的资源，所以要尽量减少内存碎片的产生。下面介绍内核获取内存的几种方式。

#### 1．伙伴算法分配大片物理内存

- alloc_pages(gfp_mask, order)：获得连续的页框，返回页描述符地址，是其他类型内存分配的基础。
- __get_free_pages(gfp_mask, order)：获得连续的页框，并返回页框对应的线性地址。线性地址与物理地址是内核直接映射方式。该方法不能用于大于 896M 的高端内存。

#### 2．Slab缓冲区分配小片物理内存

内核提供了后备高速缓存机制，称为“slab 分配器”。slab 分配器实现的高速缓存具有 kmem_cache_t 类型，可通过调用 kmem_cache_create 创建。

- kmem_cache_create：建立 slab 的高速缓冲区。
- kmem_cache_alloc：试图从本地高速缓存获得一个空闲的对象。
- kmalloc(gfp_mask, size)：获得连续的以字节为单位的物理内存，返回线性地址。

#### 3．非连续内存区分配

vmalloc(size)：分配非连续内存区，其线性地址连续，物理地址不连续，减少了外碎片，但是其性能低，因为要打乱内核页表。通常只是分配大内存时，例如为活动的交互区分配数据结构、加载内核模块时分配空间、为 I/O 驱动程序分配缓冲区。

#### 4．高端内存映射

- kmap(struct page * page)：用于获得高端内存永久内核映射的线性地址。
- kmap_atomic（struct page * page, enum km_type type)：用于获得高端内存临时内核映射的线性地址。

### 5．固定线性地址映射

- ❑ set_fixmap(idx, phys)：把一个物理地址映射到一个固定的线性地址上。
- ❑ set_fixmap_nocache(idx, phys)：把一个物理地址映射到一个固定的线性地址上，禁用该页高速缓存。

## 1.4　虚拟文件系统

虚拟文件系统的思想是把不同种类的文件系统的共同信息放入内核。其中一个字段或函数来支持 Linux 所支持的各种文件系统提供的操作。对所调用的读、写或其他函数，内核都能把它们替换成支持 Linux 文件系统、NFS 文件系统，或者其他文件系统的实际函数。在第 2 章中，会讲到 Linux 的安装，在虚拟机上安装 Linux，同时实现 Linux 和 Windows 实现文件共享，即实现在 Linux 环境下能够直接访问 Windows 的 FAT32 文件系统。

### 1.4.1　虚拟文件系统作用

虚拟文件系统（Virtual Filesystem），实际上是对各种文件系统的一种封装，为各种文件系统提供了一个通用的接口。通常情况下，为了实现不同操作系统下文件访问，例如复制/usr/local/arm 目录下的 zImage 文件到/mnt/hgfs/Windows 目录下。

```
$cp /usr/local/arm/zImage /mnt/hgfs/Windows/
```

在不同文件系统中实现文件复制，其执行的原理如图 1.2 所示。

VFS 支持的文件系统可分为以下 3 个主要类型。

#### 1．磁盘文件系统

这些文件系统管理本地磁盘中可用的存储空间或者其他可以起到磁盘作用的设备（如 USB 闪存或硬盘）。这些文件系统包括：

- ❑ Linux 使用的第二扩展文件系统（Ext2），第三扩展文件系统（Ext3）及 Reiser 文件系统（TeiserFS）。
- ❑ UNIX 家族文件系统，如 sysv 文件系统（System V、Coherent、Xenix）、UFS（BSD、Solaris、NEXTSTEP），MINIX 文件系统及 VERITAS VxFS（SCO UnixWare）。
- ❑ Windows 支持的文件系统，如 MS-DOS、FAT、FAT32、NTFS 等文件系统。
- ❑ ISO9660CD-ROM 文件系统和通用磁盘的 DVD 文件系统。

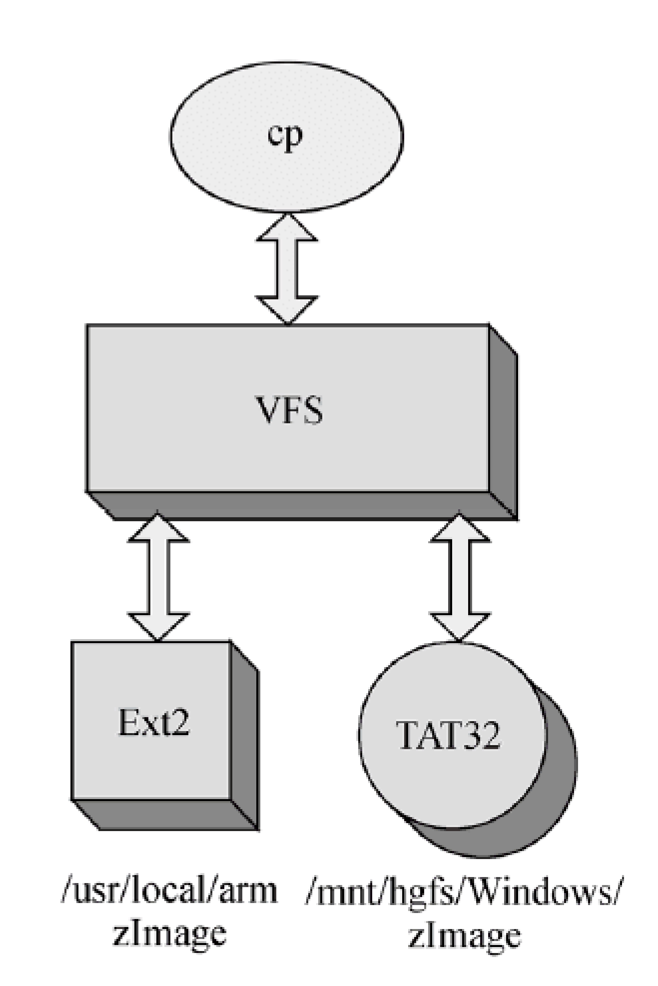



图 1.2　不同文件系统中实现文件复制

- 其他文件系统，如 HPFS（IBM 公司的 OS/2）、HFS（苹果公司的 Macintosh）、AFFS（Amiga 公司的快速文件系统）及 ADFS（Acorn 公司的磁盘文件系统）。

#### 2．网络文件系统（NFS）

网络文件系统最主要的功能就是让网络上的主机可以共享目录及资料。将远端所共享出来的系统，挂载（mount）在本地端的系统上，然后就可以很方便地使用远端的资料，而操作起来就像在本地操作一样。使用 NFS 有相当多的好处，例如文档可以集中管理、节省磁盘空间、资源共享等。

#### 3．特殊文件系统

特殊文件系统可以为系统程序员和管理员提供一种容易的方式，来操作内核的数据结构并实现操作系统的特殊特征。

### 1.4.2　文件系统的注册

每个注册的文件系统是指可能会被挂载到目录树中的各个实际文件系统。实际文件系统，即指 VFS 中的实际操作最终要通过它们来完成而已，并不表示它们必须要存在于特定的某种存储设备上。注册过程实际上是将表示各实际文件系统的 struct file_system_type 数据结构的实例化，接着形成一个链表，内核中用一个名为 file_systems 的全局变量来指向该链表的表头。下面为 file_system_type 数据结构。

```
struct file_system_type {
    const char *name;
    int fs_flags;
    int (*get_sb) (struct file_system_type *, int, const char *, void *, struct
vfsmount *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
};
```

- name：文件系统名，如 ext2。
- fs_flags：文件系统类型标志。
- get_sb：读超级块的方法。
- kill_sb：删除超级块的方法。
- owner：指向实现文件模块的文件指针。
- next：指向文件系统类型链表中下一个文件系统的指针。
- fs_supers：具有相同文件系统类型的超级块对象链表的头。

### 1.4.3　文件系统的安装和卸载

在 Linux 系统中，同一个文件系统可以被多次安装。如果一个文件系统被安装多次，

那么就可以通过这多个安装点来访问文件系统。尽管可以通过这多个安装点来访问，但是文件系统却只有一个。不论文件系统被安装了多少次，都只有一个超级块对象。安装一个文件系统遵循的步骤：

（1）Linux 系统内核必须首先检查有关参数的有效性。VFS 首先应找到准备安装的文件系统。查找的方式是，通过查找 file_systems 指针指向的链表中的 file_system_type 数据结构项，来搜索已知的文件系统（该结构中包含文件系统的名字和指向 VFS 超级块读取程序地址的指针），当找到一个匹配的名字时，就可以得到读取文件系统超级块的程序地址。

（2）查找作为新文件系统安装点的 VFS 索引结点，并且同一目录下只能安装一个文件系统；VFS 安装程序必须分配一个 VFS 超级块（super_block），并且向它传递一些有关文件系统安装的信息。

（3）申请一个 vfsmount 数据结构（其中包括存储文件系统的块设备的设备号、文件系统安装的目录和一个指向文件系统的 VFS 超级块的指针），并使它的指针指向所分配的 VFS 超级块。

（4）当文件系统安装以后，该文件系统的根索引结点就一直保存在 VFS 索引结点缓存中。

卸载文件系统：验证被卸文件系统是否为可卸载的，如果该文件系统中的文件当前正被使用，则该文件系统不能被卸载；如果文件系统中的文件或目录正在使用，则 VFS 索引节点高速缓存中可能包含对应的 VFS 索引结点；如果相应的结点标志为“被修改过”，则该文件系统不能被卸载。如果验证被卸文件系统为可卸载的，就释放相应的 VFS 超级块和安装点，从而卸载该文件系统。

vfsmount 数据结构如下：

```
struct vfsmount
{
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent;         /* fs we are mounted on */
        struct dentry *mnt_mountpoint;       /* dentry of mountpoint */
        struct dentry *mnt_root;             /* root of the mounted tree */
        struct super_block *mnt_sb;          /* pointer to superblock */
        struct list_head mnt_mounts;/* list of children, anchoredhere */
        struct list_head mnt_child; /* and going through their mnt_child */
        atomic_t mnt_count;
        int mnt_flags;
        int mnt_expiry_mark;        /* true if marked for expiry */
        char *mnt_devname;          /* Name of device e.g. /dev/dsk/hda1 */
        struct list_head mnt_list;
        struct list_head mnt_fslink;/* link in fs-specific expiry list */
        struct namespace *mnt_namespace;     /* containing namespace */
};
```

- mnt_hash：用于散列表链表的指针。
- mnt_parent：指向父文件系统，这个文件系统安装在其上。
- mnt_mountpoint：指向这个文件系统安装点目录的 dentry。
- mnt_root：指向这个文件系统根目录的 dentry。
- mnt_sb：指向这个文件系统的超级块对象。
- mnt_mounts：包含所有文件系统描述符的链表头。

- mnt_child：用于已安装文件系统 mnt_mounts 的指针。
- mnt_count：引用计数器，增加该值禁止文件系统被卸载。
- mnt_flags：安装标志。
- mnt_expiry_mark：如果文件系统到期就设置该标志为 true。
- mnt_devname：设备文件名。
- mnt_list：已安装文件描述符的 namespace 链表的指针。
- mnt_fslink：具体文件系统到期链表的指针。
- mnt_namespace：指向安装了文件系统的 namespace 链表的指针。

# 1.5　设备驱动程序

设备驱动，实际上是硬件功能的一个抽象。针对同一个硬件不同的驱动可以将硬件封装成不同的功能。设备驱动是硬件层和应用程序（或者操作系统）的媒介，能够让应用程序或者操作系统能够使用硬件。在 Linux 操作系统下有 3 类主要的设备文件类型：块设备、字符设备和网络设备。设备驱动程序是指管理某个外围设备的一段代码，它负责传送数据、控制特定类型的物理设备的操作，包括开始和完成 I/O 操作，检测和处理设备出现的错误。

## 1.5.1　字符设备驱动程序

字符设备是一种能像字节流一样进行串行访问的设备，对设备的存取只能按顺序按字节存取而不能随机访问，字符设备没有请求缓冲区，必须按顺序执行所有的访问请求。应用程序对字符设备的访问是通过字符设备结点来完成的。字符设备是 Linux 中最简单的设备，可以像文件一样访问。应用程序使用标准系统调用打开、读、写和关闭字符设备，完全可以把它们当作普通文件一样进行操作，甚至被 PPP 守护进程使用，用于将一个 Linux 系统连接到网上的 modem，也被看作一个普通文件。当字符设备初始化时，它的设备驱动程序向 Linux 内核注册，向 chrdevs 向量表中增加一个 device_struct 数据结构项。通常一种类型设备的主设备标识符是固定的。设备的主设备标识符（例如对于 tty 设备是 4），用作该向量表的索引。chrdevs 向量表中的每一项，即 device_struct 数据结构，包括两个元素：一个是指向登记的设备驱动程序名字的指针；另一个是指向一组文件操作的指针。这组文件操作本身位于这个设备的字符设备驱动程序中，每一个都处理一个特定的文件操作，如打开、读、写和关闭。常见的字符设备有鼠标、键盘、串口、控制台等。

用户进程通过设备文件对硬件进行访问，对设备文件的操作方式通过一些系统调用来实现，如 open、read、write 和 close 等。下面通过一个关键的数据结构 file_operations，将系统调用和驱动程序关联起来。

```
struct file operations {
    int  (*seek)  (struct inode * , struct file *,  off_t , int);
    int  (*read)  (struct inode * , struct file *,  char , int);
    int  (*write)  (struct inode * , struct file *,  off_t , int);
    int  (*readdir)  (struct inode * , struct file *,  struct dirent * , int);
```

```
    int (*select) (struct inode * , struct file *,  int , select_table *);
    int (*ioctl) (struct inode * , struct file *,  unsined int , unsigned long);
    int (*mmap) (struct inode * , struct file *,  struct vm_area_struct *);
    int (*open) (struct inode * , struct file *);
    int (*release) (struct inode * , struct file *);
    int (*fsync) (struct inode * , struct file *);
    int (*fasync) (struct inode * , struct file *, int);
    int (*check_media_change) (struct inode * , struct file *);
    int (*revalidate) (dev_t dev);
};
```

该结构中每一个成员的名字都对应着一个系统调用。用户进程利用系统调用在对设备文件进行诸如 read/write 操作时，系统调用根据设备文件的主设备号找到对应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数。

编写驱动程序就是针对上面相应的函数编写具体的实现，然后将它们对应上。编写完驱动后，把驱动程序嵌入内核。驱动程序可以采用两种方式进行编译。一种是编译进内核，驱动被静态加载；另一种是编译成模块（modules），驱动模块需要动态加载。在模块被调入内存时，init()函数向系统的字符设备表登记了一个字符设备

```
int __init chr_dev_init(void)
{
    if (devfs register chrdev(CHR MAJOR,"chr name",&chr fops))
        printk("unable to get major %d for chr devs\n", MEM_MAJOR);
    …
    return 0;
}
```

当 cleanup_chr_dev()函数被调用时，它释放字符设备 chr_name 在系统字符设备表中占有的表项。

```
void cleanup_chr_dev(void)
{
    unregister_chrdev(CHR_MAJOR, "chr_name");
}
```

## 1.5.2　块设备驱动程序

块设备具有请求缓冲区，从块设备读取数据时，可以从任意位置读取任意长度，即块设备支持随机访问而不必按照顺序存取数据。例如，可以先存取后面的数据，然后再存取前面的数据，字符设备则不能采用该方式存取数据。Linux 下的磁盘设备均为块设备，应用程序访问 Linux 下的块设备结点是通过文件系统及其高速缓存来访问块设备的，并非直接通过设备结点读写块设备上的数据。

块设备既可以用作普通的裸设备存放任意数据，也可以将块设备按某种文件系统类型的格式进行格式化，然后读取块设备上的数据，读取时根据该文件系统类型的格式进行读取。无论使用哪种方式，访问设备上的数据都必须通过调用设备本身的操作方法实现。两者区别在于前者直接调用块设备的操作方法，而后者则间接调用块设备的操作方法。常见的块设备有各种硬盘、flash 磁盘、RAM 磁盘等。

块设备也可以与字符设备 register_chrdev、unregister_chrdev()函数类似的方法进行设备的注册与释放。但是，字符设备的 register_chrdev()函数使用一个 file_operations 结构的指针，而块设备的 register_blkdev()函数则使用 block_device_operations 结构的指针，其中定义的 open、release 和 ioctl 方法和字符设备的对应方法相同，但没有对 read 和 write 操作定义。这是因为所有涉及到块设备的 I/O 通常由系统进行缓冲处理。

块驱动程序最终必须提供完成实际块 I/O 操作的机制，在 Linux 中，用于这些 I/O 操作的方法称为 request（请求）。注册块设备时，需要对 request 队列进行初始化，这一动作通过 blk_init_queue 来完成，blk_init_queue 函数创建队列，并将该驱动程序的 request 函数关联到队列。在模块的清除阶段，应调用 blk_cleanup_queue 函数。

初始化块设备的时候，将块设备注册到内核中，下面为块设备的注册函数 mtdblock_release()的实现：

```
int register_blkdev(unsigned int major, const char *name)
{
    struct blk_major_name **n, *p;
    int index, ret = 0;
    mutex_lock(&block_class_lock);
    /*为块设备指定主设备号，如果指定为 0 则表示由系统来分配*/
    if (major == 0) {
        for (index = ARRAY_SIZE(major_names)-1; index > 0; index--) {
            if (major names[index] == NULL)
                break;
        }
        if (index == 0) {
            printk("register_blkdev: failed to get major for %s\n",
                   name);
            ret = -EBUSY;
            goto out;
        }
        major = index;
        ret = major;
    }
    /*为块设备名字分配空间*/
    p = kmalloc(sizeof(struct blk_major_name), GFP_KERNEL);
    if (p == NULL) {
        ret = -ENOMEM;
        goto out;
    }
    p->major = major;
    strlcpy(p->name, name, sizeof(p->name));
    p->next = NULL;
    index = major_to_index(major);
    for (n = &major_names[index]; *n; n = &(*n)->next) {
        if ((*n)->major == major)
            break;
    }
    if (!*n)
        *n = p;
    else
        ret = -EBUSY;
    if (ret < 0) {
        printk("register_blkdev: cannot get major %d for %s\n",
              major, name);
        kfree(p);
    }
```

```
out:
    mutex_unlock(&block_class_lock);
    return ret;
}
```

块设备被注册到系统后，访问硬件的操作 open 和 release 等就能够被对应的系统调用指针所绑定，应用程序使用系统调用就可以对硬件进行访问了，下面是块设备主要的操作函数 open()和 release()。

下面为块设备 open()操作函数。

```
static int mtdblock_open(struct mtd_blktrans_dev *mbd)
{
    struct mtdblk_dev *mtdblk;
    struct mtd_info *mtd = mbd->mtd;
    int dev = mbd->devnum;
    DEBUG(MTD DEBUG LEVEL1,"mtdblock open\n");
    if (mtdblks[dev]) {
        /*如果设备已经打开，则只需要增加其引用计数*/
        mtdblks[dev]->count++;
        return 0;
    }
    /*为设备创建 mtdblk_dev 对象保存 mtd 设备的信息*/
    mtdblk = kzalloc(sizeof(struct mtdblk dev), GFP KERNEL);
    if (!mtdblk)
        return -ENOMEM;
    mtdblk->count = 1;
    mtdblk->mtd = mtd;
    mutex init(&mtdblk->cache mutex);
    mtdblk->cache_state = STATE_EMPTY;
    if ( !(mtdblk->mtd->flags & MTD_NO_ERASE) && mtdblk->mtd->erasesize) {
        mtdblk->cache size = mtdblk->mtd->erasesize;
        mtdblk->cache_data = NULL;
    }
    mtdblks[dev] = mtdblk;
    DEBUG(MTD_DEBUG_LEVEL1, "ok\n");
    return 0;
}
```

释放时递减用户计数，当用户计数递减为 0 时，释放缓存中的数据并释放为设备分配的空间。

```
static int mtdblock_release(struct mtd_blktrans_dev *mbd)
{
    int dev = mbd->devnum;
    struct mtdblk_dev *mtdblk = mtdblks[dev];
    DEBUG(MTD DEBUG LEVEL1, "mtdblock release\n");
    mutex_lock(&mtdblk->cache_mutex);
    write cached data(mtdblk);
    mutex_unlock(&mtdblk->cache_mutex);
    if (!--mtdblk->count) {
        /*用户计数递减为 0 时释放设备*/
        mtdblks[dev] = NULL;
        if (mtdblk->mtd->sync)
            mtdblk->mtd->sync(mtdblk->mtd);
        vfree(mtdblk->cache_data);
        kfree(mtdblk);
    }
    DEBUG(MTD_DEBUG_LEVEL1, "ok\n");
```

```
    return 0;
}
```

### 1.5.3　网络设备驱动程序

网络设备与字符设备的区别是，网络设备是面向数据报文的，而字符设备是面向字符流的。网络设备与块设备的区别是，网络设备不支持随机访问，也没有请求缓冲区。在 Linux 里网络设备也可以被称为网络接口，如 eth0，应用程序是通过 Socket（套接字），而不是设备结点来访问网络设备，在系统中不存在网络设备结点。

网络设备用来与其他设备交换数据，它可以是硬件设备，也可以是纯软件设备，如 loopback 接口。网络设备由内核中的网络子系统驱动，负责发送和接收数据包，但它不需要了解每项事务如何映射到实际传送的数据包。许多网络连接（例如使用 TCP 协议的连接）是面向流的，但网络设备围绕数据包的传输和接收设计。网络驱动程序不需要知道各个连接的相关信息，它只需处理数据包。字符设备和块设备都有设备号，而网络设备没有设备号，只有一个独一无二的名字，例如 eth0、eth1 等，这个名字也无须与设备文件结点对应。内核利用一组数据包传输函数与网络设备驱动程序进行通信，它们不同于字符设备和块设备的 read()和 write()方法。

Linux 网络设备驱动程序从下到上分为 4 层，依次为网络设备与媒介层、设备驱动功能层、网络设备接口层和网络协议接口层。在设计具体的网络设备驱动程序时，需要完成的主要工作是编写设备驱动功能层的相关函数以填充 net_device 数据结构的内容并将 net_device 注册入内核。

下面以 DM9000 代码为例说明网络设备驱动的注册、注销等主要过程。其驱动的注册过程在设备初始化时被调用。

```
static int __init dm9000_init(void)
{
    printk(KERN_INFO "%s Ethernet Driver, V%s\n", CARDNAME, DRV_VERSION);
    return platform_driver_register(&dm9000_driver);
}
```

驱动注册函数。

```
int platform_driver_register(struct platform_driver *drv)
{
    drv->driver.bus = &platform_bus_type;
    if (drv->probe)
        drv->driver.probe = platform_drv_probe;
    if (drv->remove)
        drv->driver.remove = platform_drv_remove;
    if (drv->shutdown)
        drv->driver.shutdown = platform_drv_shutdown;
    if (drv->suspend)
        drv->driver.suspend = platform_drv_suspend;
    if (drv->resume)
        drv->driver.resume = platform_drv_resume;
    return driver_register(&drv->driver);
}
```

在网络设备被清除时调用注销网络设备驱动函数。

```
static void __exit dm9000_cleanup(void)
{
    platform_driver_unregister(&dm9000_driver);
}
```

驱动注销过程。驱动注销的过程中还包括了将设备从系统中移除和将驱动从总线上移植。

```
void platform_driver_unregister(struct platform_driver *drv)
{
    driver_unregister(&drv->driver);
}
void driver_unregister(struct device_driver *drv)
{
    driver_remove_groups(drv, drv->groups);
    bus_remove_driver(drv);
}
static void driver_remove_groups(struct device_driver *drv, struct
attribute_group **groups)
{
    int i;
    if (groups)
        for (i = 0; groups[i]; i++)
            sysfs_remove_group(&drv->p->kobj, groups[i]);
}
void bus_remove_driver(struct device_driver *drv)
{
    if (!drv->bus)
        return;
    remove_bind_files(drv);
    driver_remove_attrs(drv->bus, drv);
    driver_remove_file(drv, &driver_attr_uevent);
    klist_remove(&drv->p->knode_bus);
    pr_debug("bus: '%s': remove driver %s\n", drv->bus->name, drv->name);
    driver_detach(drv);
    module_remove_driver(drv);
    kobject_put(&drv->p->kobj);
    bus_put(drv->bus);
}
```

有关网络设备驱动的详细接口函数解析和驱动移植将放在后面的章节叙述。

### 1.5.4　内存与 I/O 操作

一般来说，在系统运行时，外设的 I/O 内存资源的物理地址是已知的，由硬件的设计决定。但是 CPU 通常并没有为这些已知的外设 I/O 内存资源的物理地址，预定义虚拟地址范围，驱动程序并不能直接通过物理地址访问 I/O 内存资源，只能先将它们映射到内核的虚拟地址空间内（通过页表），然后才能根据映射的内核虚拟地址范围，通过访内指令访问这些 I/O 内存资源。Linux 在 io.h 头文件中声明了函数 ioremap()，用来将 I/O 内存资源的物理地址映射到核心虚拟地址空间（3GB－4GB）中，原型如下：

```
void * ioremap(unsigned long phys_addr, unsigned long size, unsigned long
flags);
```

iounmap()函数用于取消 ioremap()所做的映射，原型如下：

```
void iounmap(void * addr);
```

在将 I/O 内存资源的物理地址映射成内核的虚拟地址后，理论上可以像读写 RAM 那样直接读写 I/O 内存资源了。为了保证驱动程序跨平台的可移植性，应该使用 Linux 中特定的函数访问 I/O 内存资源，而不应该通过指向内核虚拟地址的指针来访问。如在 ARM 平台上，读写 I/O 的函数如下：

```
#define __raw_base_writeb(val,base,off)  __arch_base_putb(val,base,off)
#define __raw_base_writew(val,base,off)  __arch_base_putw(val,base,off)
#define __raw_base_writel(val,base,off)  __arch_base_putl(val,base,off)

#define __raw_base_readb(base,off)  __arch_base_getb(base,off)
#define __raw_base_readw(base,off)  __arch_base_getw(base,off)
#define __raw_base_readl(base,off)  __arch_base_getl(base,off)
```

驱动程序中 mmap()函数的实现原理是，用 mmap 映射一个设备，表示将用户空间的一段地址关联到设备内存上，这样当程序在分配的地址范围内进行读取或者写入时，实际上就是对设备的访问。这一映射原理类似于 Linux 下 mount 命令，将一种类型的文件系统或设备挂载到另外一个文件系统或者目录下时，挂载成功后，对挂载点的任何操作实际上是对被挂载的文件系统和设备的操作。

## 1.6 小　　结

Linux 内核是一个比较庞大的系统，深入理解内核可以减少在系统移植中的障碍。在系统移植中设备驱动开发是一项很复杂的工作，由于 Linux 内核提供了相当部分的源代码，同时还提供了对某些公共部分的支持。例如，USB 驱动对读写 U 盘、键盘、鼠标等设备提供了通用驱动程序。一般情况可以直接使用内核提供的驱动，但是对于复杂的 USB 设备没有现成的驱动，就需要读者对驱动开发过程有一定的认识，必要时参考 Linux 源码重新开发驱动程序。

# 第 2 章　嵌入式 Linux 开发环境搭建

进行嵌入式项目开发，需要建立嵌入式开发环境。建立嵌入式 Linux 开发环境包括安装 Bootloader 工具；针对不同平台的交叉编译器（在本书中都是针对 ARM 平台）arm-linux-gcc；需要编译配置内核时还需要安装内核源码树；此外还要在调试时使用一些终端软件、TFTP 软件、FTP 软件。还有内核和文件系统的烧写工具，一般硬件厂家会提供这样的工具。本章主要介绍嵌入式 Linux 系统移植过程中用到的交叉编译环境建立，以及各种工具的安装和配置。

## 2.1　虚拟机及 Linux 安装

很多工具都是 Windows 版本的，而要求的开发环境是 Linux 环境。在 Windows 系统中安装虚拟机，然后再虚拟一个 Linux 环境，使 Linux 和 Windows 能够互相通信。这种方案解决了很多软件不兼容两种平台的问题。

### 2.1.1　虚拟机的安装

虚拟机软件 Vmware 的安装和普通软件安装过程一样，就不详细介绍了。这里主要介绍在虚拟机中安装 Linux 系统的过程，强调一个关键的地方。正确安装 VMware 后启动时的界面如图 2.1 所示。

下面介绍安装 Linux 的主要步骤，省略了一些只需要单击“下一步”按钮的步骤。

（1）准备安装系统软件 FC-6-i386-DVD.iso，可以在网上下载。

（2）运行 VMware，选择“文件”|“新建”|“虚拟机”命令，或者直接单击“新建虚拟机”图标。

（3）在虚拟机配置窗口中选择“自定义”选项，如图 2.2 所示。

（4）虚拟机硬件兼容性窗口按默认选择安装。

（5）在选择客户机操作系统的时候选择 Linux 选项。在“版本”下拉菜单中选择准备安装的 Linux 版本。这里使用的是默认的软件版本，如果不是 Red Hat Linux 可以选择其他版本，如图 2.3 所示。

（6）单击“下一步”按钮，进入主题为虚拟机取名称的对话框。在此对话框中为安装的虚拟机取名字，同时确定安装路径，注意选择安装的分区应该有足够的空间安装 Linux 系统。因为在后面还要安装 Linux 源码树，所以建议安装在一个有 8～10GB 空闲空间的分区上。

（7）单击“下一步”按钮，进入处理器配置对话框，在其中根据实际情况选择处理器

的个数。




图 2.1　VMware 启动界面

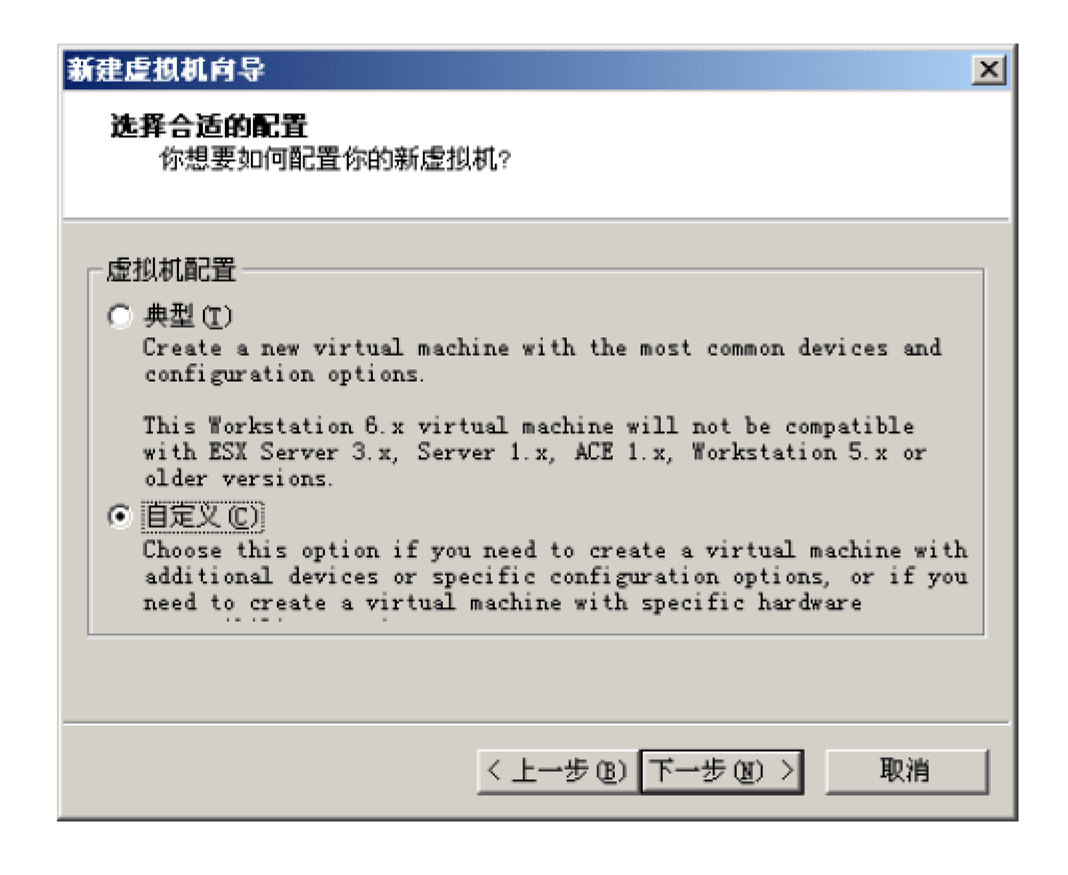



图 2.2　虚拟机配置

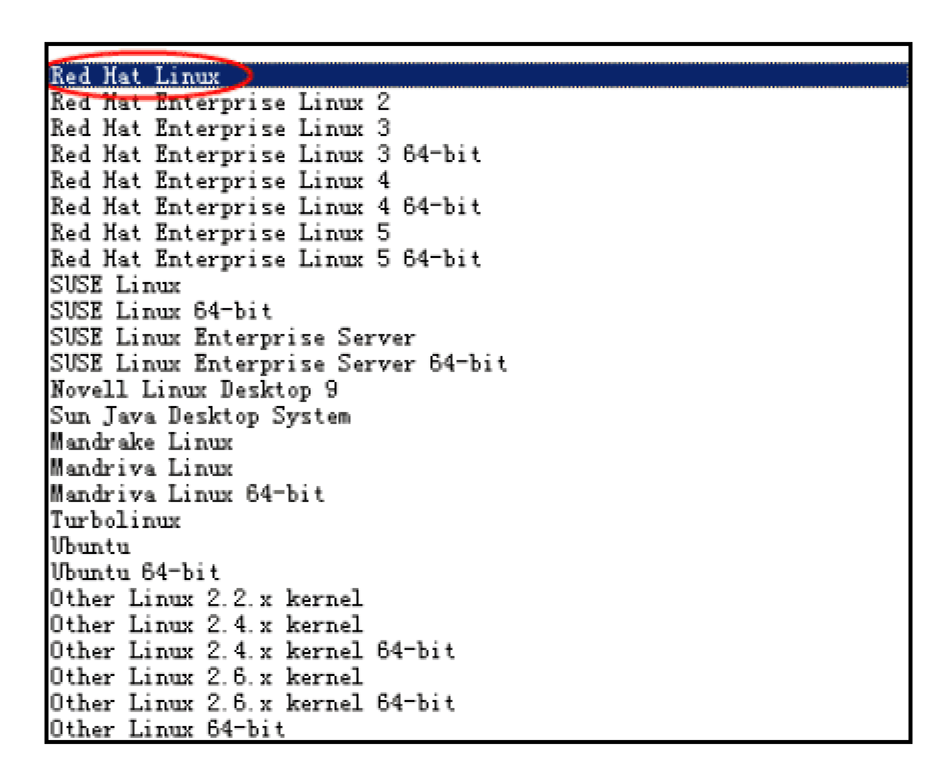



图 2.3　Linux 安装版本选择

（8）在对虚拟机内存进行划分时，可以根据实际主机硬件的配置进行划分，一般可以按默认的配置安装。如果实际主机配置比较高，可以给虚拟机多分配点内存。

（9）在网络连接类型中可以选择任意一种类型，该设置在后面需要修改的时候可以进行修改，此处可以按默认选项进行安装。

（10）I/O 适配器窗口可以按照默认配置安装。

（11）在“选择一个磁盘”对话框中，选择“创建一个新的虚拟磁盘”单选按钮，如图 2.4 所示。

（12）在“选择磁盘类型”对话框中，选择 IDE 单选按钮，如图 2.5 所示。

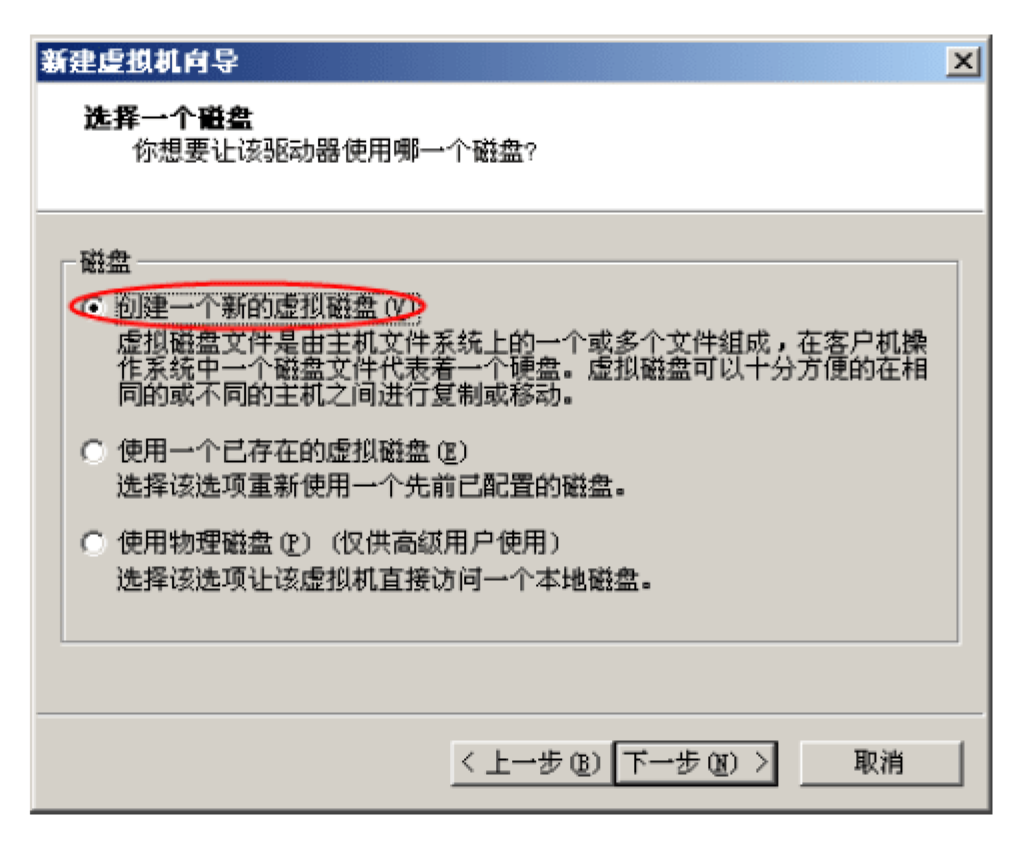



图 2.4　选择创建一个新的虚拟磁盘

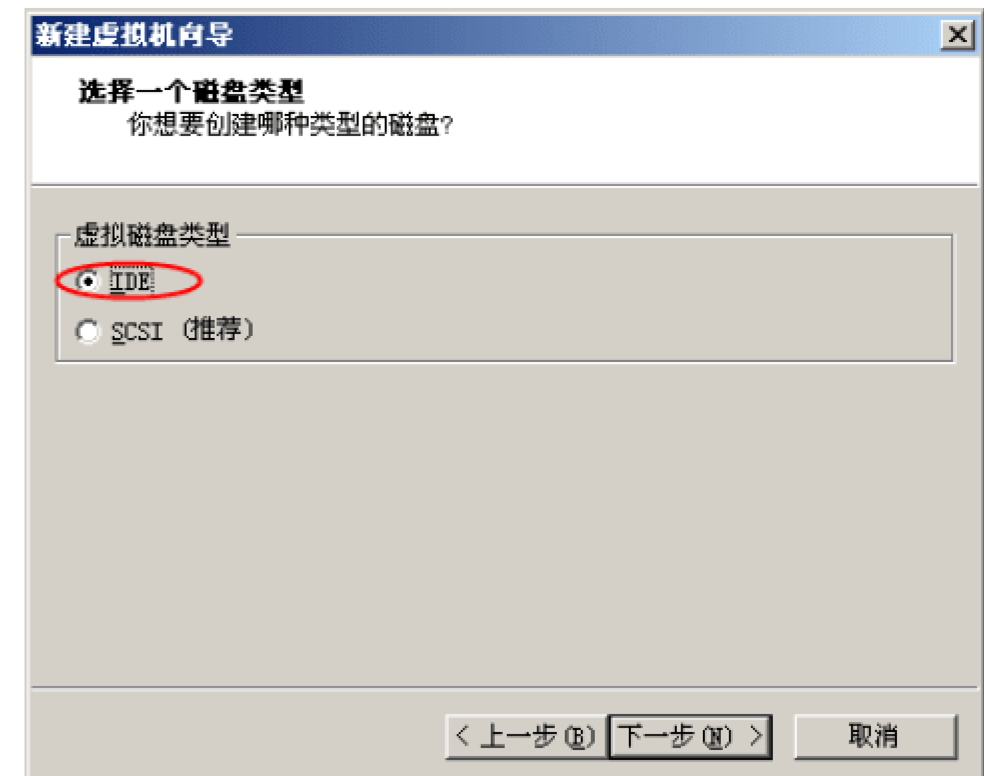



图 2.5　虚拟磁盘类型

（13）在指定磁盘容量对话框中，确定磁盘大小，为 Linux 系统预留 8～10GB 空间，而且勾选“立即分配所有磁盘空间”复选框，如图 2.6 所示。

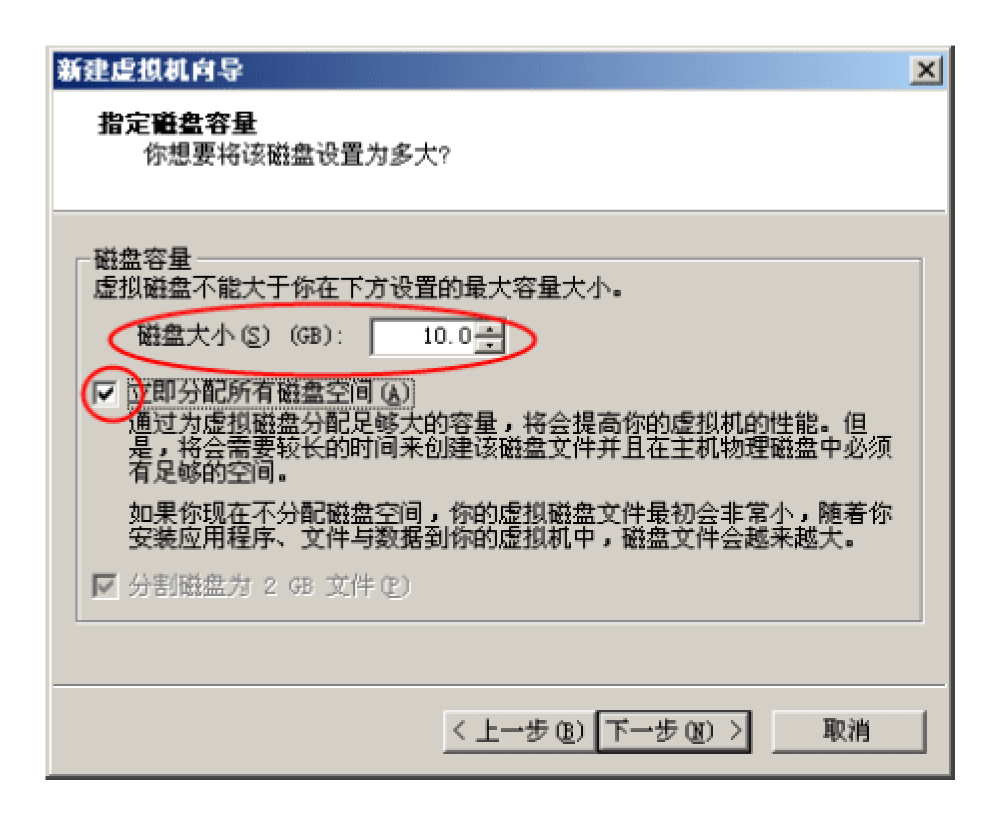



图 2.6　指定磁盘容量

（14）指定磁盘文件对话框按默认名字和路径进行安装，单击“完成”按钮，开始创建磁盘。创建完成后，结果如图 2.7 所示。

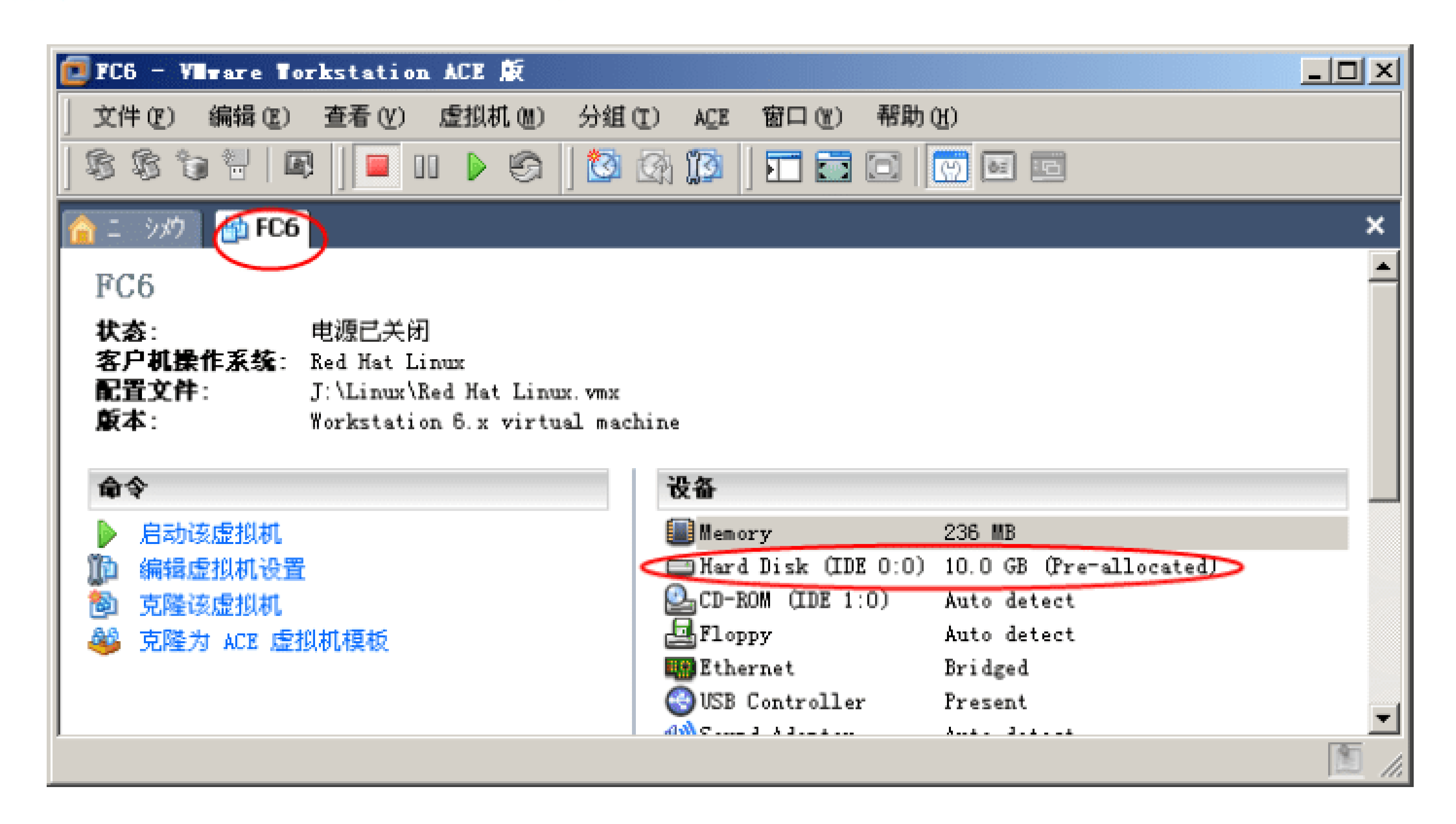



图 2.7　创建虚拟机

（15）安装虚拟光驱软件，将 Linux 安装软件放入虚拟光驱。

（16）找到虚拟光驱在 Windows 中的指定盘符，如图 2.8 所示。将虚拟机 CD-ROM 指定为刚才看到的盘符，如图 2.9 所示，然后单击 OK 按钮。

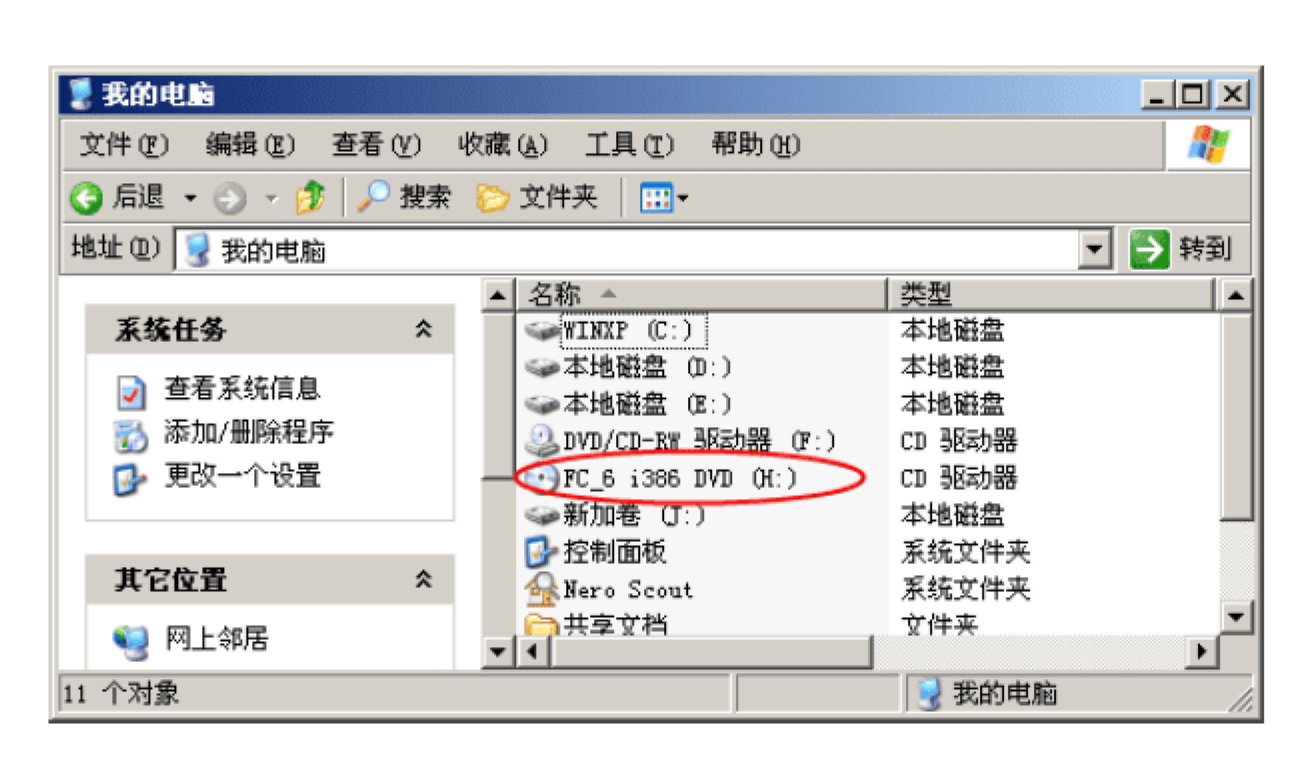



图 2.8　虚拟光驱在 Windows 中对应的盘符

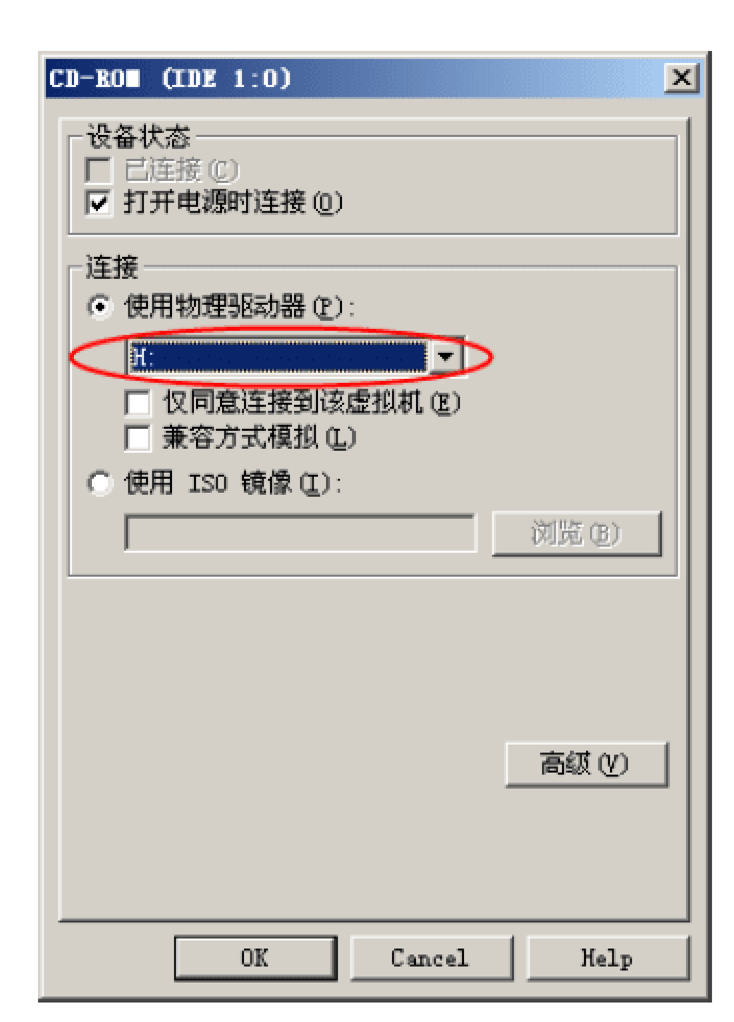



图 2.9　指定虚拟机中光驱

（17）在工具栏中选择“启动该虚拟机”，或者在“命令”区域选择“启动该虚拟机”选项，如图 2.10 所示。

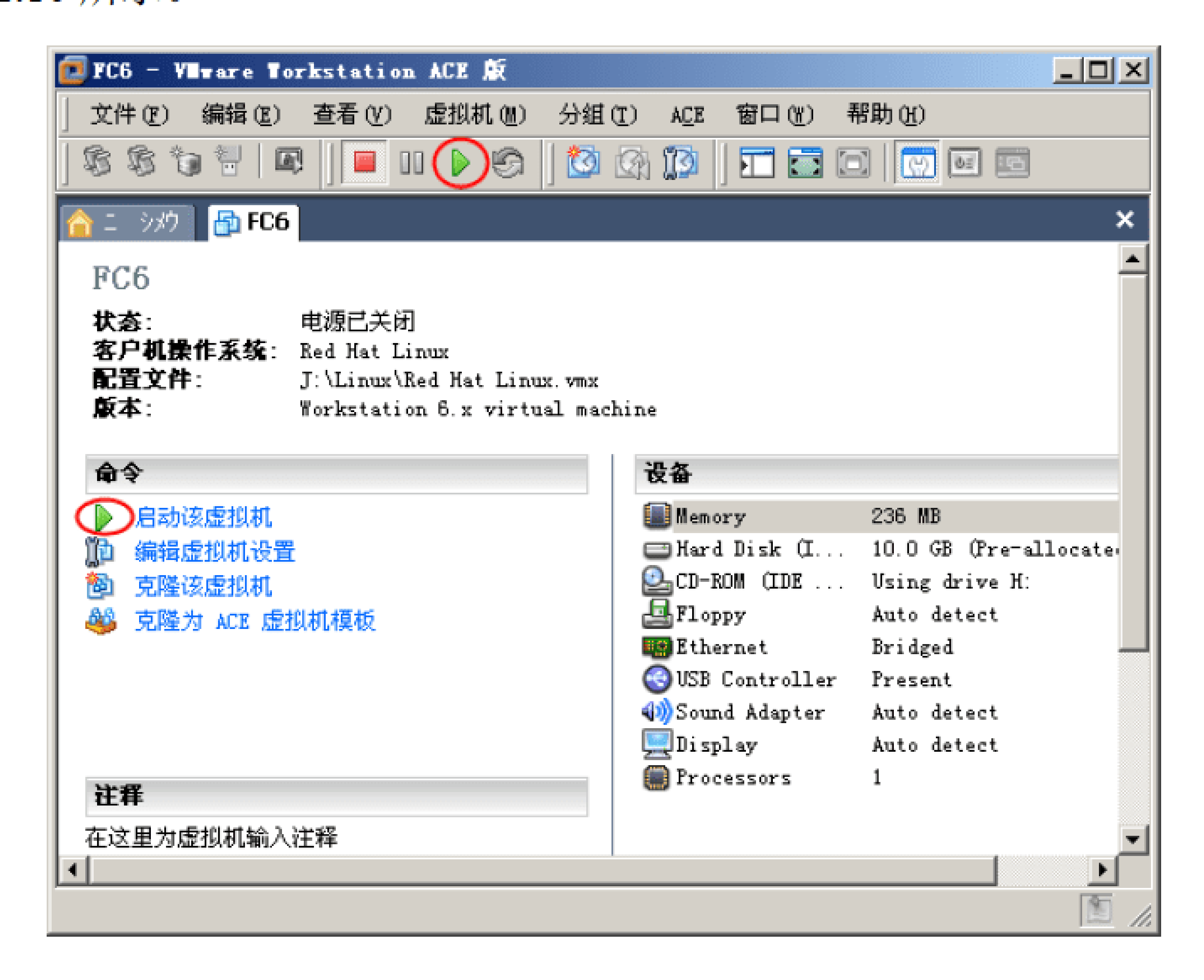



图 2.10　启动虚拟机

（18）进入虚拟机安装界面，如图 2.11 所示。按下回车键，开始安装系统，如图 2.12 所示。

（19）在测试 CD 选项对话框中可以选择 OK 按钮或者 Skip 按钮。

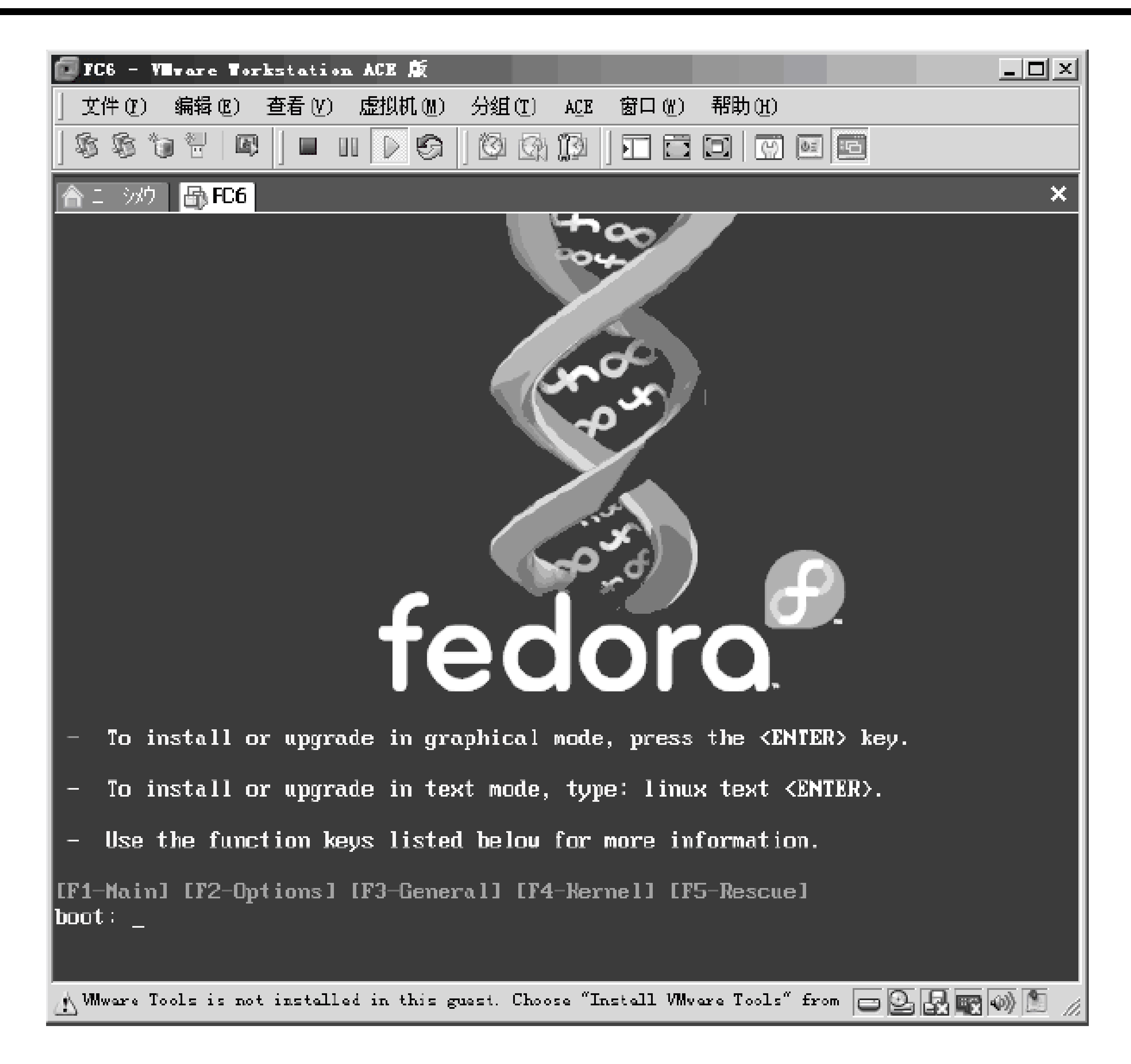



图 2.11　进入安装界面

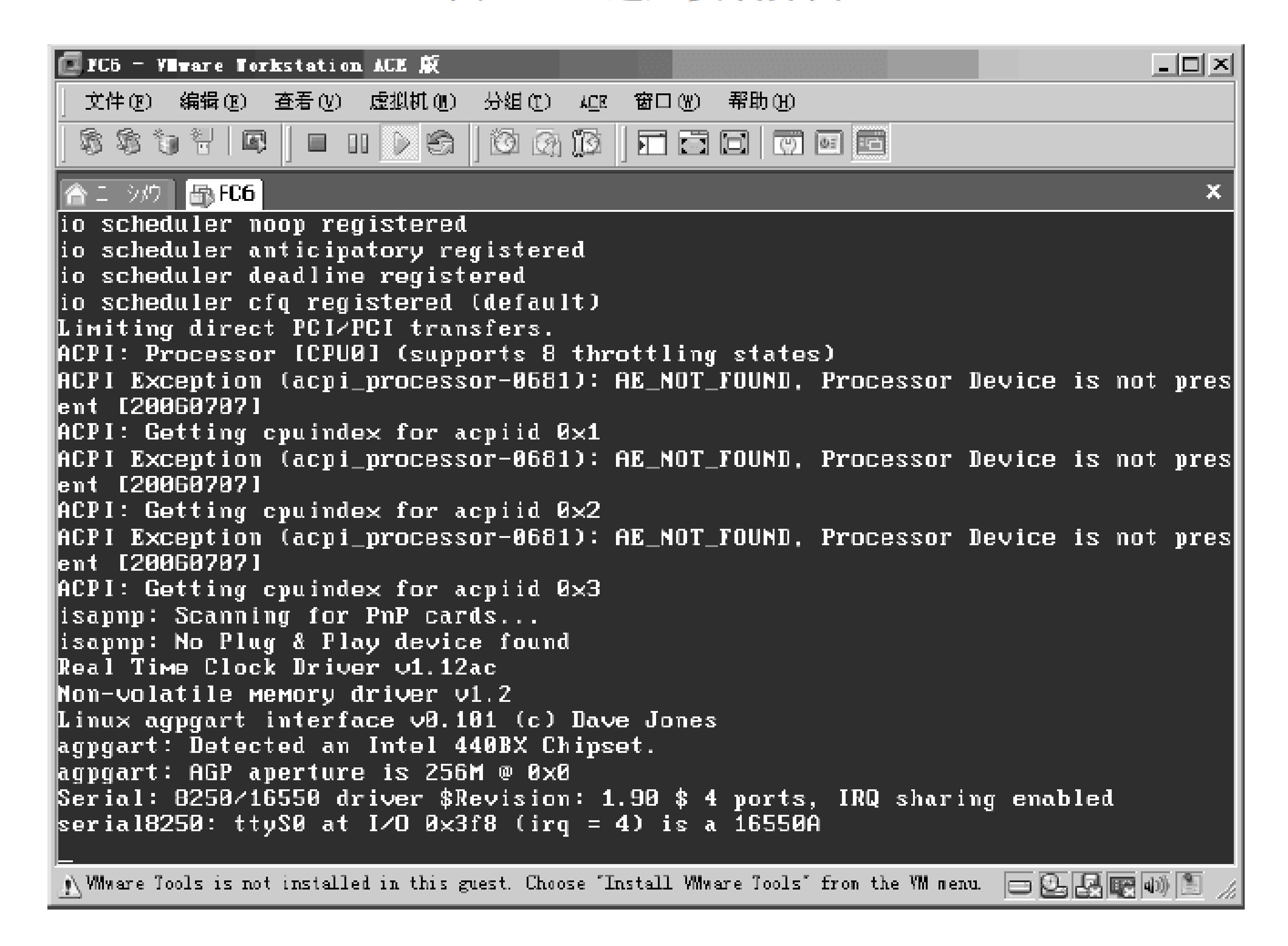



图 2.12　安装开始界面

（20）在选择语音的时候可以选择“简体中文”，当然也可以选择 English 或者其他任何你喜欢的语音。英文的选择对话框也可以选“美国英语式”或者其他你喜欢的方式。这些选项也可以在系统安装完成后进行修改。之后几步都是单击“下一步”按钮进行默认安装。

（21）Linux 系统密码不能为空，而且至少为 6 位密码，设置密码后继续单击“下一步”按钮。

（22）在进入的对话框中选择安装额外的软件包时，可以根据实际情况选择。这里选择“软件开发”复选框，系统自带开发工具，如图 2.13 所示，单击“下一步”按钮继续安装。




图 2.13　选择需要额外安装的软件包

（23）系统安装完成后重新引导系统，进入配置界面时，将 FTP、NFS4、Samba、Telnet、HTTP 服务都启用，如图 2.14 所示。这些服务也可以在进入系统以后进行设置。




图 2.14　添加服务

（24）安装完成后的界面如图 2.15 所示。

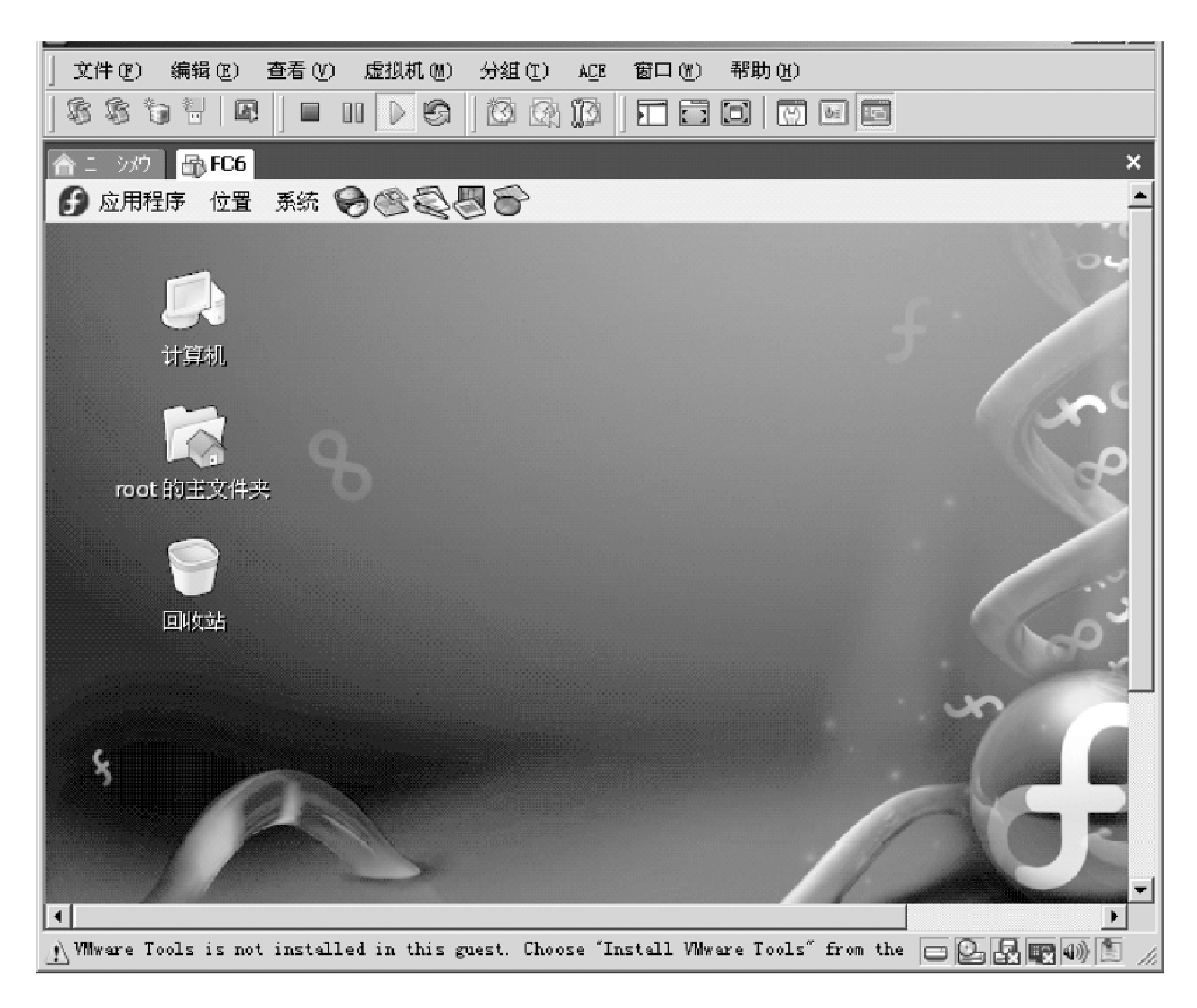



图 2.15　完成安装 Linux 系统

注意：安装过程中的几个关键地方有磁盘分区的大小；采用虚拟光驱安装，要将系统软件放在虚拟光驱中，并且在该虚拟机中进行设置；硬盘类型选择 IDE；指定磁盘容量时要选“立即分配磁盘空间”。这几步是安装的关键，如果选择错误将导致系统无法正确安装。

注意：因为没有安装 VMware tools 工具，鼠标还不能在虚拟机和主机之间进行切换，那么需要同时按下 Ctrl+Alt 键释放鼠标。

## 2.1.2　单独分区安装系统

在某些开发的场合需要将 Linux 单独安装在磁盘的某个分区上，此时需要注意设置两点。其一，虚拟机的 CD/DVD 设备状态设置为设置启动时连接，设置方法如图 2.16 所示；其二，在虚拟机的 BIOS 中设置第一启动为 CD-ROM，在启动虚拟机时按下 F2 键进入虚拟机的 BIOS 设置界面，如图 2.17 所示。

设置完成后，按 F10 键保存设置并重新启动进行安装，安装过程与前面的方法基本相同。另外，分区的大小最好大于 10GB，笔者在分区小于 10GB 情况下安装 Fedora 8 时失败，如果要编译 x86 内核，则分区大小至少 15GB 以上。

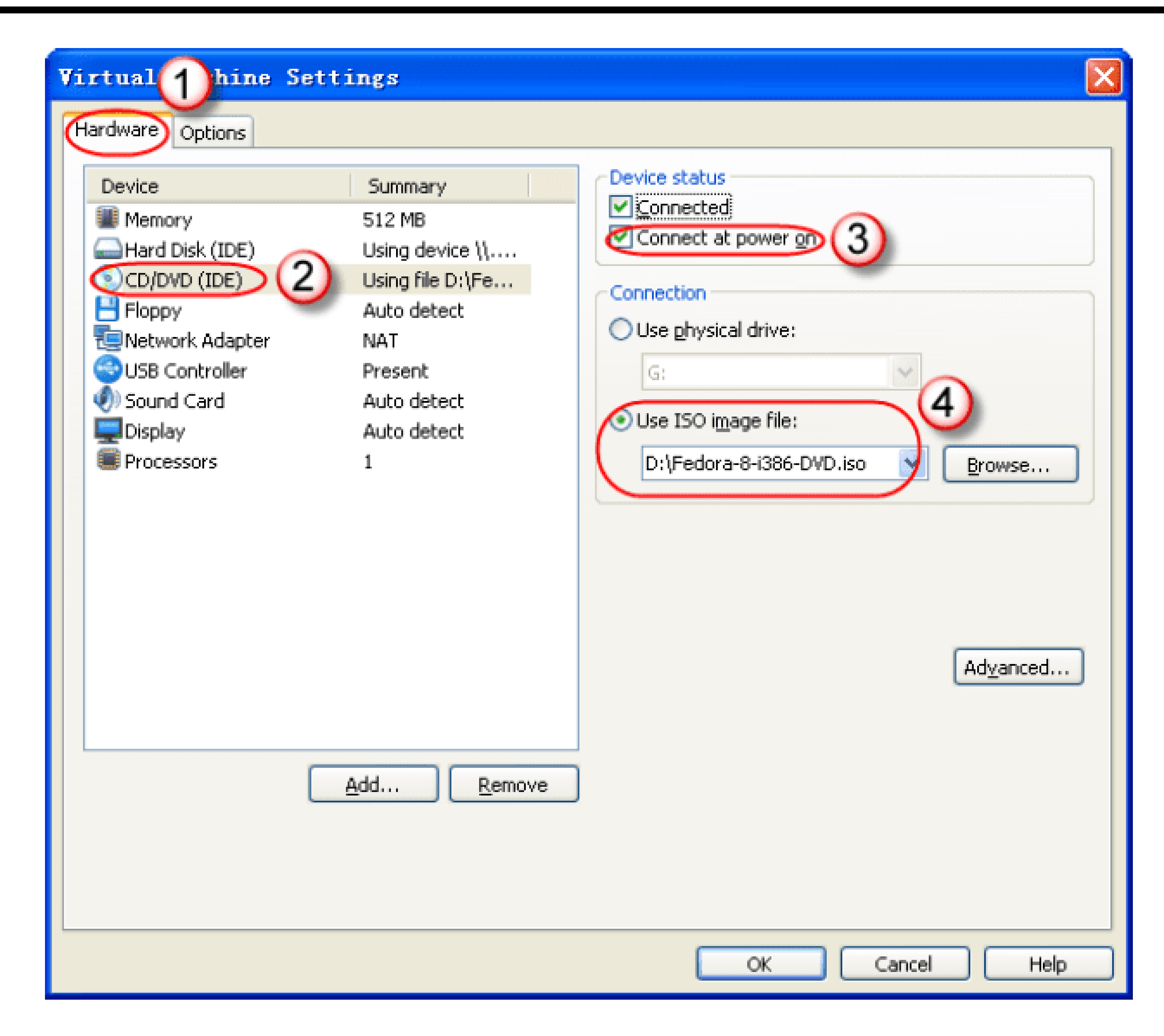



图 2.16　设置启动虚拟机时连接光驱

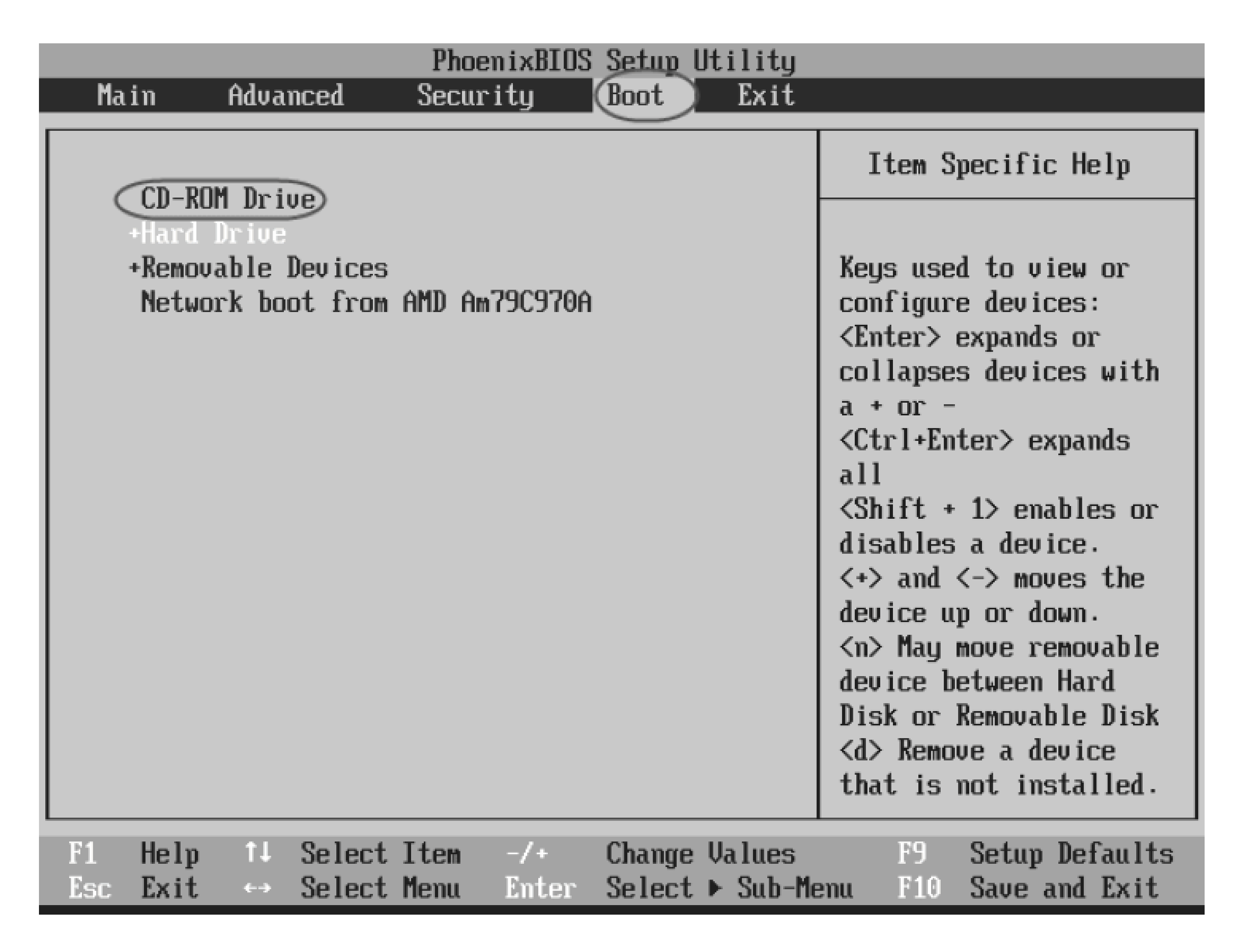



图 2.17　设置虚拟机 BIOS 启动选项

## 2.1.3　虚拟机和主机通信设置

很多资料和软件往往都放在主机上，当需要在虚拟机环境下对这些资料进行访问时，或者虚拟机编译好的文件传送到主机上时，就需要建立两者之间的通信。建立虚拟机和主机通信的过程如下：

（1）选择“虚拟机”|“设置”命令，打开虚拟机设置对话框。选择 Hardware 标签中的 Ethernet 选项。在“网络连接”选项区域中选择“自定义（s）：指定虚拟网络”单选按钮，在下拉列表框中设置网络连接为 VMnet8 (NAT)，单击 OK 按钮，如图 2.18 所示。

（2）选择“编辑”|“虚拟网络设置”命令，进入虚拟网络编辑对话框。选择 NAT 标签，将 VMnet host 设置为 VMnet8。然后单击“确定”按钮，如图 2.19 所示。

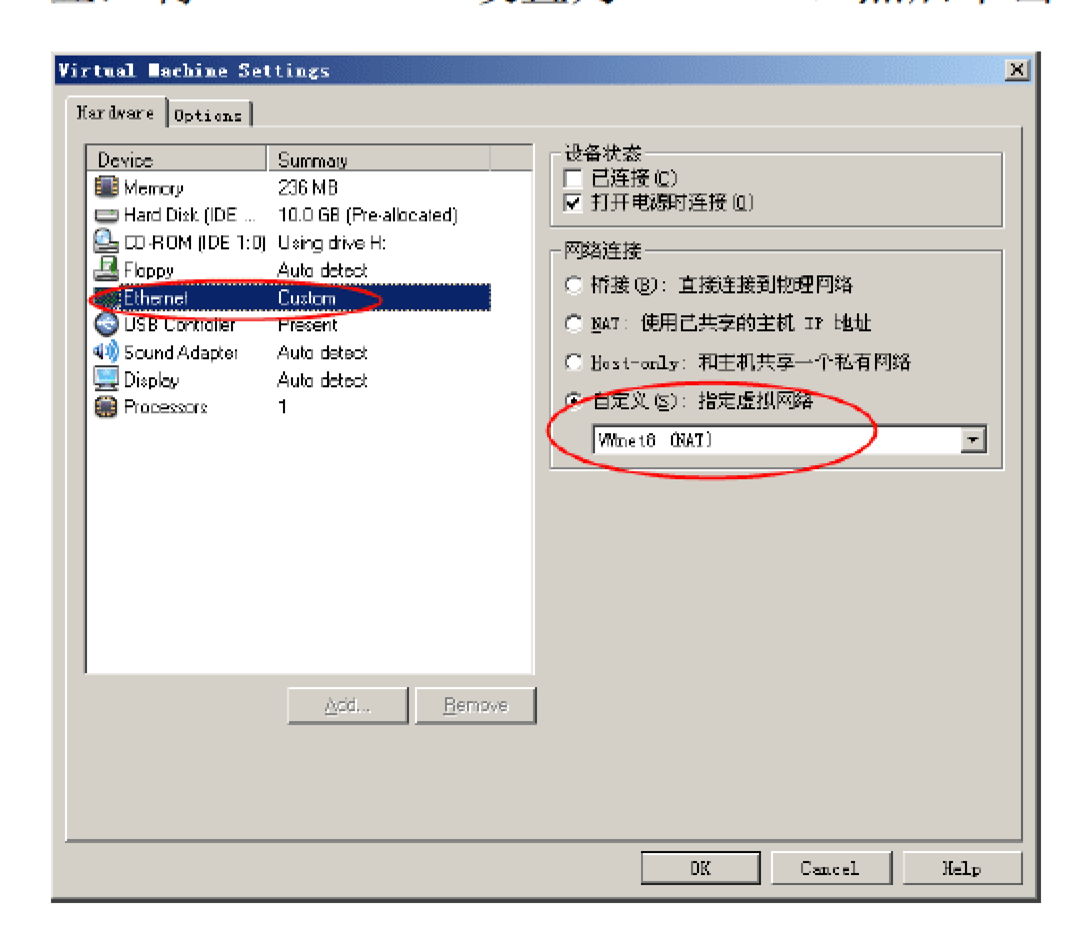



图 2.18　设置网络连接

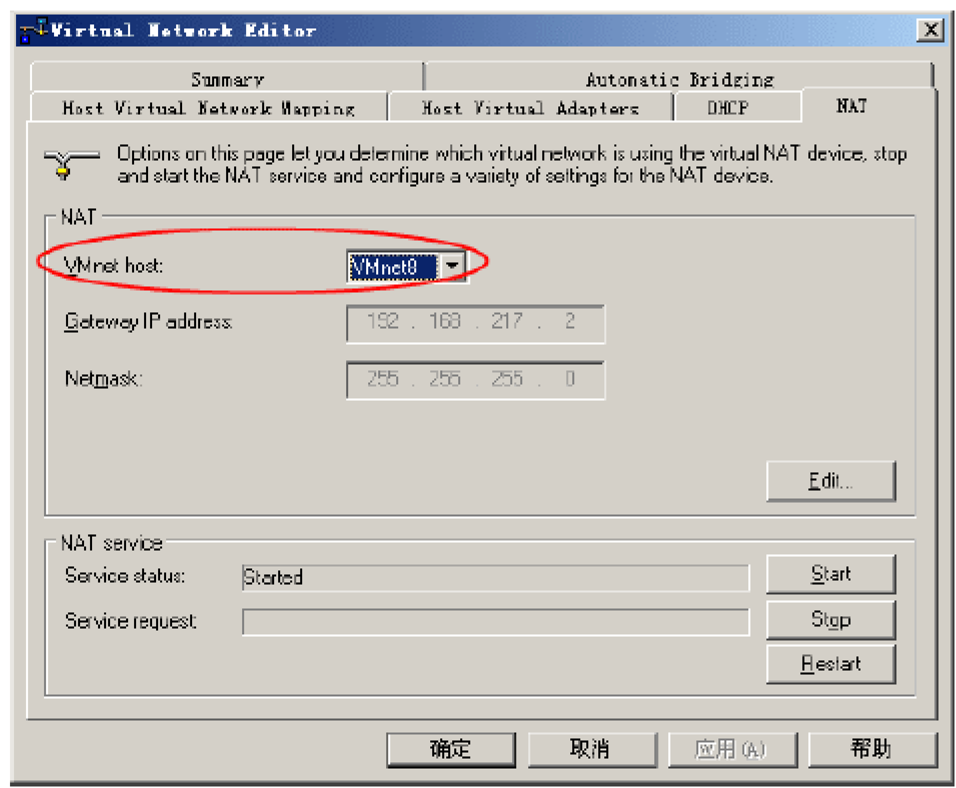



图 2.19　设置虚拟网络

（3）设置网卡连接状态。双击虚拟机窗口右下角以太网标志，如图 2.20 所示。进入以太网设置对话框后，选择“已连接”和“打开电源时连接”复选框，如图 2.21 所示。




图 2.20　双击网卡标志

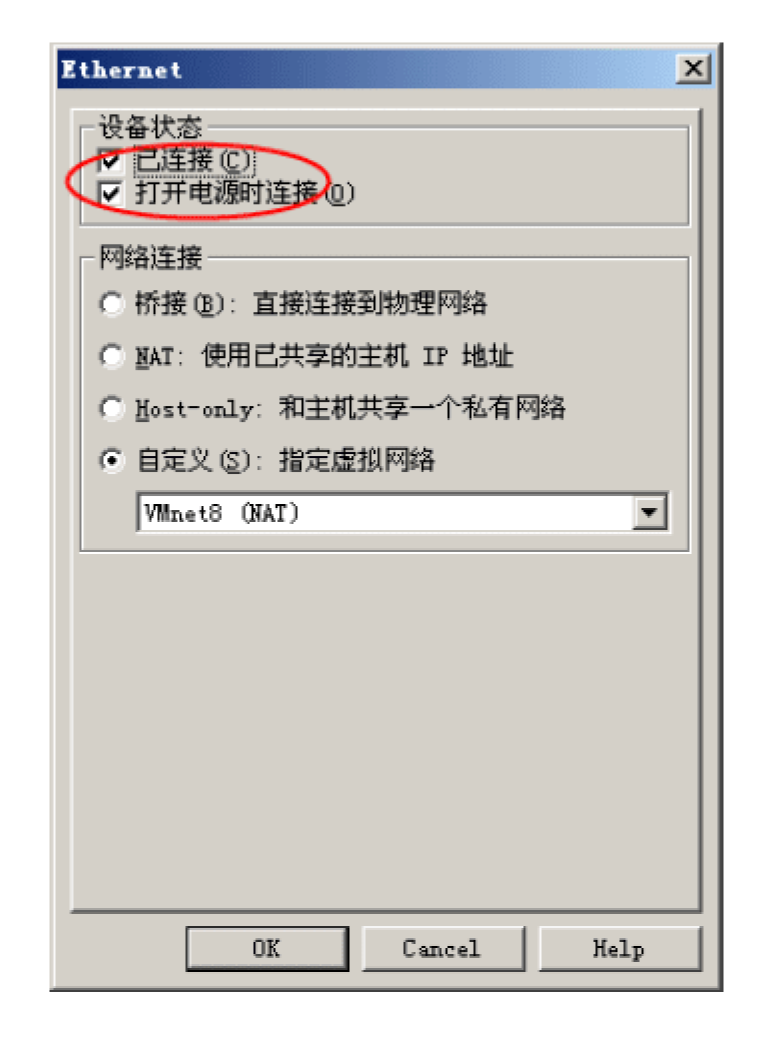



图 2.21　设置网卡状态

（4）打开一个终端，通过 ifconfig 查看虚拟机的 IP 地址。如果没有对网卡 IP 设置，则使用 ifconfig eth0 192.168.217.55。注意这里的 IP 与第二步网关的 IP 段设置为同一个网段。没有设置 IP 地址时，查看的信息如图 2.22 所示。设置 IP 地址后查看 IP 地址时，如图 2.23 所示。

```
#ifconfig
```

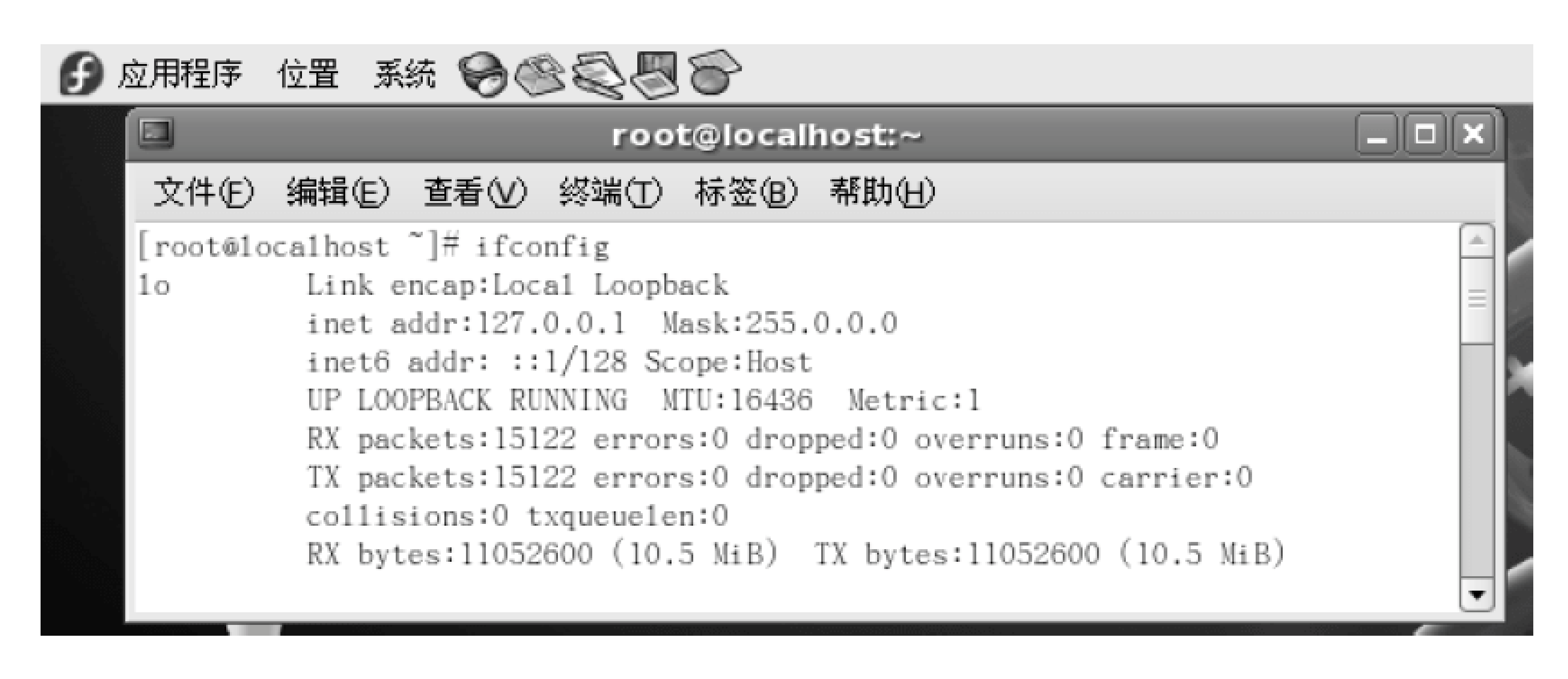



图 2.22　设置 IP 地址前

```
#ifconfig eth0 192.168.217.55
#ifconfig
```

```
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:57:4F:4A
          inet addr:192.168.217.55  Bcast:192.168.217.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe57:4f4a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:26 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:5630 (5.4 KiB)
          Interrupt:169 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:15122 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15122 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:11052600 (10.5 MiB)  TX bytes:11052600 (10.5 MiB)

[root@localhost ~]# █
```

图 2.23　设置 IP 地址后

（5）确定主机的 VMware net8 网卡为已连接状态，在主机端 ping 虚拟机，结果如图 2.24 所示。查看主机 VMware net8 网卡的 IP 地址，一般和虚拟机网关为一个网段，并且其 IP 地址为 192.168.217.1。在虚拟机中 ping 该 IP，结果如图 2.25 所示。互相都可以 ping 通说明主机和虚拟机通信成功。

## 2.1.4　VMware tools 工具安装

没有安装 VMware tools 工具，要在主机和虚拟机之间进行切换很不方便，并且也不能进行复制、粘贴命令的操作。在 2.1.3 节建立了主机和虚拟机通信的基础上，可以安装

VMware Tools 工具。安装 VMware tools 工具后为使用提供了很多方便。

```
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>ping 192.168.217.55

Pinging 192.168.217.55 with 32 bytes of data:

Reply from 192.168.217.55: bytes=32 time=2ms TTL=64
Reply from 192.168.217.55: bytes=32 time<1ms TTL=64
Reply from 192.168.217.55: bytes=32 time<1ms TTL=64
Reply from 192.168.217.55: bytes=32 time<1ms TTL=64

Ping statistics for 192.168.217.55:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 2ms, Average = 0ms

C:\Documents and Settings\Administrator>
```

图 2.24　主机 ping 虚拟机

```
          RX bytes:0 (0.0 b)  TX bytes:5630 (5.4 KiB)
          Interrupt:169 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:15122 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15122 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:11052600 (10.5 MiB)  TX bytes:11052600 (10.5 MiB)

[root@localhost ~]# ping 192.168.217.1
PING 192.168.217.1 (192.168.217.1) 56(84) bytes of data.
64 bytes from 192.168.217.1: icmp_seq=1 ttl=128 time=1.39 ms
64 bytes from 192.168.217.1: icmp_seq=2 ttl=128 time=0.245 ms
64 bytes from 192.168.217.1: icmp_seq=3 ttl=128 time=0.132 ms
64 bytes from 192.168.217.1: icmp_seq=4 ttl=128 time=0.221 ms
64 bytes from 192.168.217.1: icmp_seq=5 ttl=128 time=0.136 ms
64 bytes from 192.168.217.1: icmp_seq=6 ttl=128 time=0.222 ms
64 bytes from 192.168.217.1: icmp_seq=7 ttl=128 time=0.135 ms
64 bytes from 192.168.217.1: icmp_seq=8 ttl=128 time=0.212 ms
64 bytes from 192.168.217.1: icmp_seq=9 ttl=128 time=0.135 ms
64 bytes from 192.168.217.1: icmp_seq=10 ttl=128 time=0.229 ms
64 bytes from 192.168.217.1: icmp_seq=11 ttl=128 time=0.141 ms
64 bytes from 192.168.217.1: icmp_seq=12 ttl=128 time=0.221 ms
64 bytes from 192.168.217.1: icmp_seq=13 ttl=128 time=0.132 ms
64 bytes from 192.168.217.1: icmp_seq=14 ttl=128 time=0.281 ms
64 bytes from 192.168.217.1: icmp_seq=15 ttl=128 time=0.139 ms
```

图 2.25　虚拟机 ping 主机

（1）以 root 身份进入 Linux 系统，然后选择“虚拟机”|“安装 VMware Tools 工具”命令，单击命令后则会变成“取消安装 VMware Tools 工具”，表示已经正确选择了“安装 VMware Tools 工具”命令，如图 2.26 所示。

（2）使用光驱加载镜像文件。双击虚拟机右下角光驱标志，打开 CD-ROM 对话框。在“使用 ISO 镜像”中添加 VMware 安装目录下的 linux.iso 文件，如图 2.27 所示。

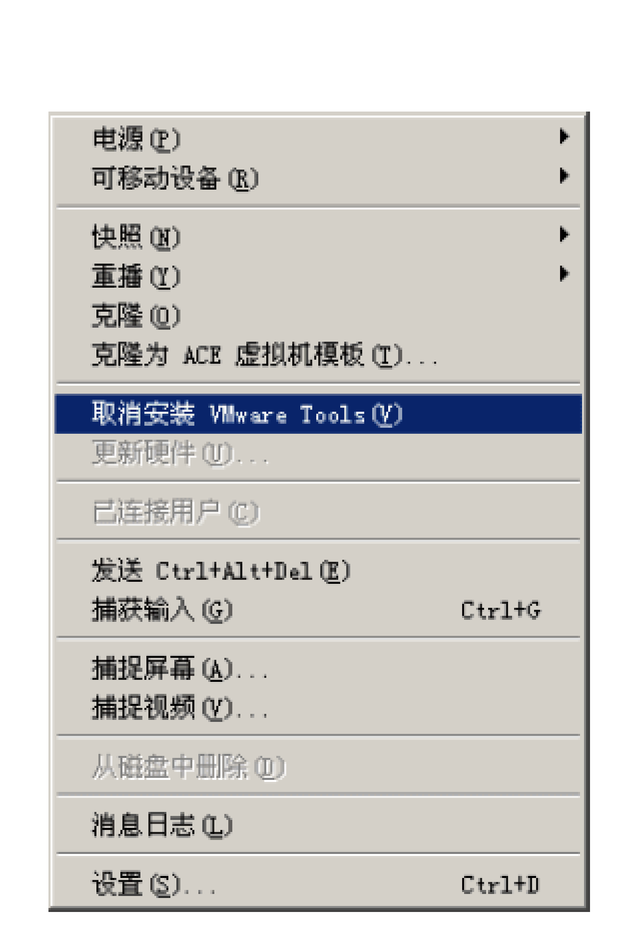



图 2.26　单击安装 VMware Tools 工具

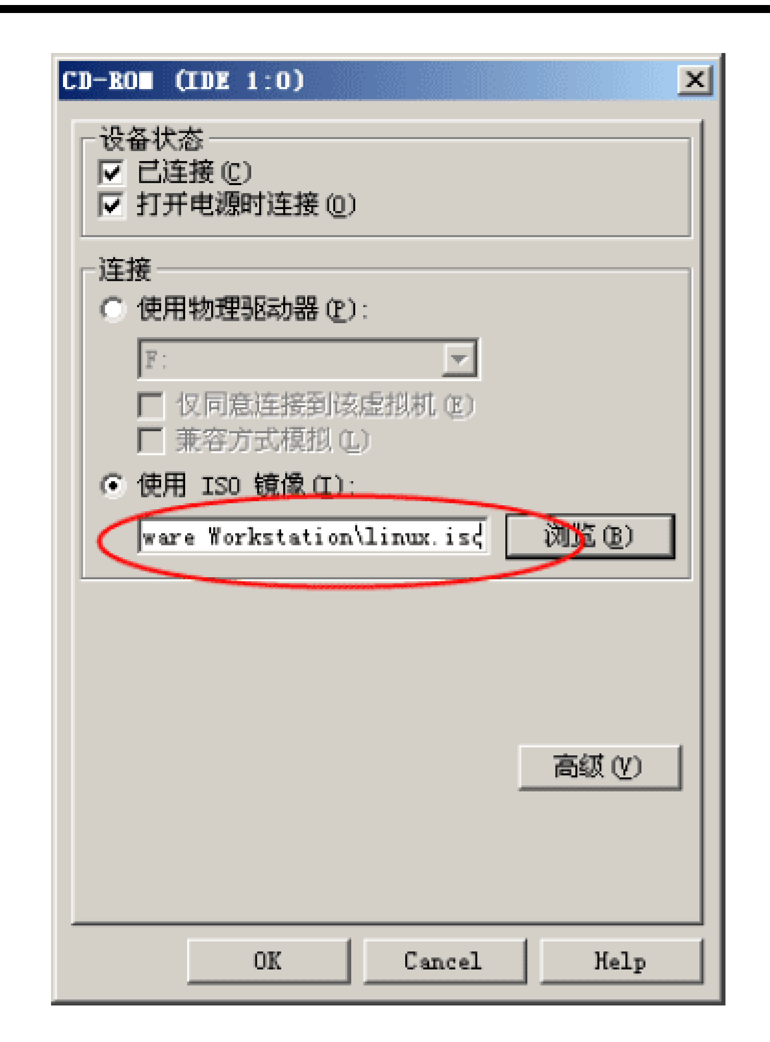



图 2.27　虚拟光驱中添加 linux.iso 安装路径

（3）在/mnt 目录下新建 cdrom 目录；将虚拟光驱中的文件挂载到该目录下，查看该目录下会存在两个文件，一个是 WMware Tools 的.rpm 文件，另一个是 VMware Tools 的.tar.gz 文件。

```
#mkdir /mnt/cdrom
#mount -t iso9660 /dev/cdrom /mnt/cdrom
#ls /mnt/cdrom -l
```

（4）将安装文件复制到/tmp 目录下，安装 rpm 包，并解压该.tar.gz 文件，进入解压目录安装 vmware-install.pl。

```
#cp /mnt/cdrom/*                                    /复制安装文件到 tmp 目录下
#cd /tmp                                            //进入 tmp 目录
#rpm -ivh VMwareTools-7.8.5-156735i386.rpm          //安装 rpm 包
#tar -zxvf VMwareTools-6.0.2-59824.tar.gz           //解压文件
#cd vmware-tools-distrib             //进入解压生成的 vmware-tools-distrib 目录
#./*.pl            //运行当前目录中后缀为 pl 的文件，即运行 vmware-install.pl
```

注意：在接下来的安装过程中，可以全部按照默认的选项进行选择，直到最后选择分辨率。安装完成后，鼠标可以在主机和虚拟机上复制文件、任意切换。

（5）安装完成后，卸载光驱，卸载命令如下：

```
#umount/mnt/cdrom
```

## 2.1.5　虚拟机与主机共享文件

设置文件共享后，能够在主机和虚拟机之间进行文件传输。

（1）选择“虚拟机”|“设置”命令打开虚拟机设置（Virtual Machine Setting）对话框。选择 Options（选项）标签，在其中选择共享文件夹。在文件夹共享选项组中选择“总是启用”单选按钮，如图 2.28 所示。

（2）单击“添加”按钮，设置主机的共享路径和共享名称，如图 2.29 所示。

（3）设置共享属性为启用，然后单击“确定”按钮。进入/mnt 目录下，会发现多了一个目录 hgfs。进入 hgfs，可以看到在 Windows 系统下的文件。显示信息如图 2.30 所示。

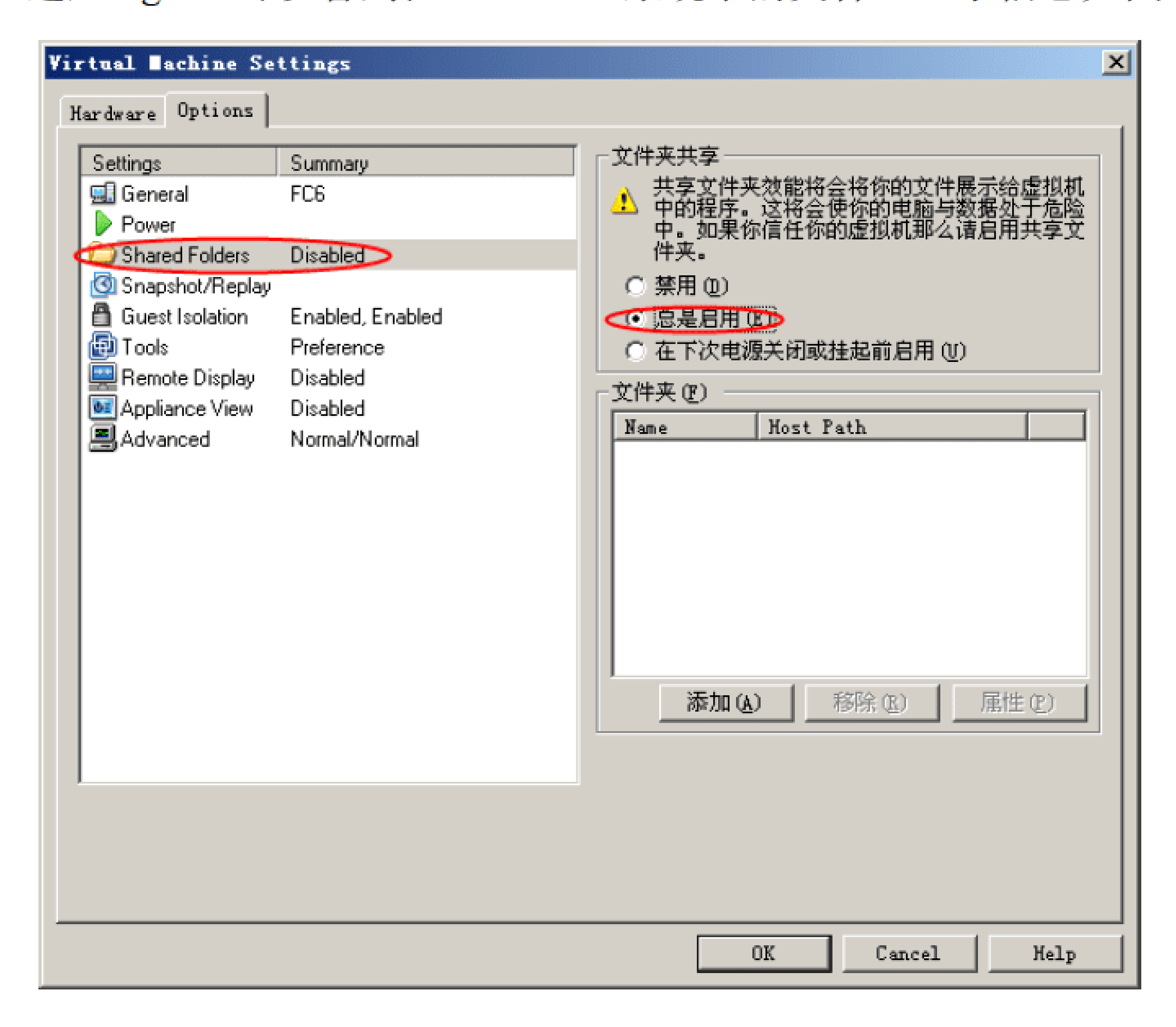



图 2.28　添加启用共享

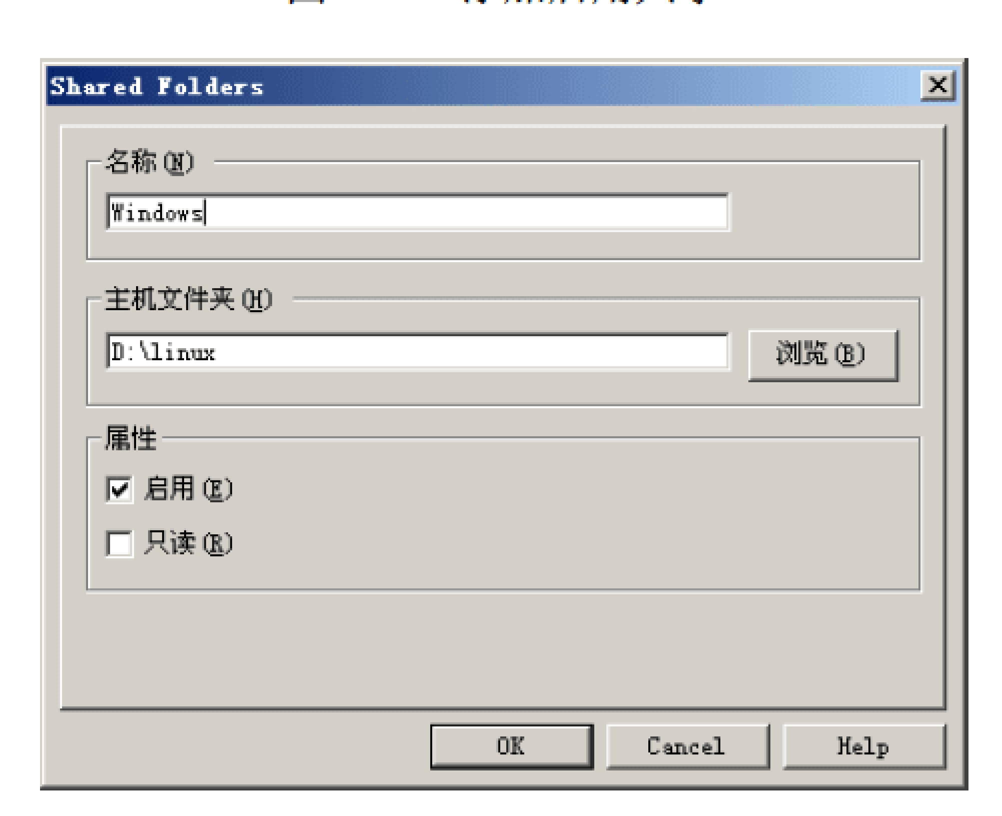



图 2.29　设置主机的共享目录

## 2.1.6　虚拟机与主机文件传输

某些版本的虚拟机或者 Linux 系统对文件共享支持不够完美，此时可以选择 FTP 方式进行文件传输，该方法操作方便，在实际开发中被普遍使用。该方法包括服务器端（虚拟机）和客户端（主机）两部分安装，并且包括服务器端和客户端的配置。下面分别进行介绍：

（1）服务器端安装 vsftpd 服务，该服务可以在安装操作系统时安装。如果没有安装，可以重新挂载安装盘进行安装。如果正确安装 vsftpd 服务，则可以在系统服务（System

Services）中选择该服务。在终端输入 setup 命令进入 System Services 进行配置，如图 2.31 所示。

```
[root@localhost /]# ls /mnt/hgfs/Windons/ -l
总计 148185
-rwxrwxrwx 1 root root 88158593 2009-03-30 arm-linux-gcc-4.3.2.tgz
drwxrwxrwx 1 root root        0 10-24 22:22 boa
-rwxrwxrwx 1 root root  2057886 11-29 17:37 crosstool-0.27-gcc3.4.1.tar.gz
drwxrwxrwx 1 root root        0 12-10 00:56 dnw
drwxrwxrwx 1 root root        0 12-28 18:19 lib
-rwxrwxrwx 1 root root   389343 12-25 21:04 libusb-0.1.12.tar.gz
-rwxrwxrwx 1 root root   322956 12-23 22:10 libusb-1.0.6.tar.bz2
drwxrwxrwx 1 root root        0 11-27 20:14 linux-2.4.34
drwxrwxrwx 1 root root        0 12-07 21:06 linux-2.6.29
drwxrwxrwx 1 root root        0 12-17 10:18 linux-2.6.29.6
-rwxrwxrwx 1 root root 56550359 12-07 23:07 linux-2.6.29.6.tar.bz2
-rwxrwxrwx 1 root root 74590693 11-12 02:21 linux-2.6.29-HY2440.tgz
-rwxrwxrwx 1 root root 64424138 12-07 23:17 linux-2.6.32.tar.bz2
drwxrwxrwx 1 root root        0 2008-01-17 LINUX系统分析与高级编程技术
drwxrwxrwx 1 root root        0 10-24 22:22 madplay-pkg
-rwxrwxrwx 1 root root    33394 01-06 00:45 option.ko
drwxrwxrwx 1 root root        0 12-28 18:19 ppp
-rwxrwxrwx 1 root root    49543 12-26 09:42 ppp-2.4.3-mppe-mppc-1.1.patch.gz
-rwxrwxrwx 1 root root   688092 12-26 09:42 ppp-2.4.3.tar.gz
-rwxrwxrwx 1 root root   603110 12-26 10:30 pppd_old
drwxrwxrwx 1 root root        0 12-27 21:08 pppon_
-rwxrwxrwx 1 root root    18664 10-25 23:17 tftp-0.39-1.i386.rpm
drwxrwxrwx 1 root root        0 2004-09-15 tftp-hpa-0.39
-rwxrwxrwx 1 root root   102139 10-25 23:47 tftp-hpa-0.39.tar.gz
-rwxrwxrwx 1 root root    25108 10-25 23:45 tftp-server-0.39-1.i386.rpm
-rwxrwxrwx 1 root root    32240 12-25 23:55 usb_modeswitch
-rwxrwxrwx 1 root root    40172 12-21 23:12 usb_modeswitch-1.0.5.tar.bz2
-rwxrwxrwx 1 root root    28914 12-25 23:58 usb_modeswitch.conf
-rwxrwxrwx 1 root root    33539 01-08 18:50 usbserial.ko
```

图 2.30　显示共享目录信息

（2）服务端必须关闭防火墙。在系统服务（System Services）中关闭 ip6tables 和 iptables 服务，如图 2.32 所示。

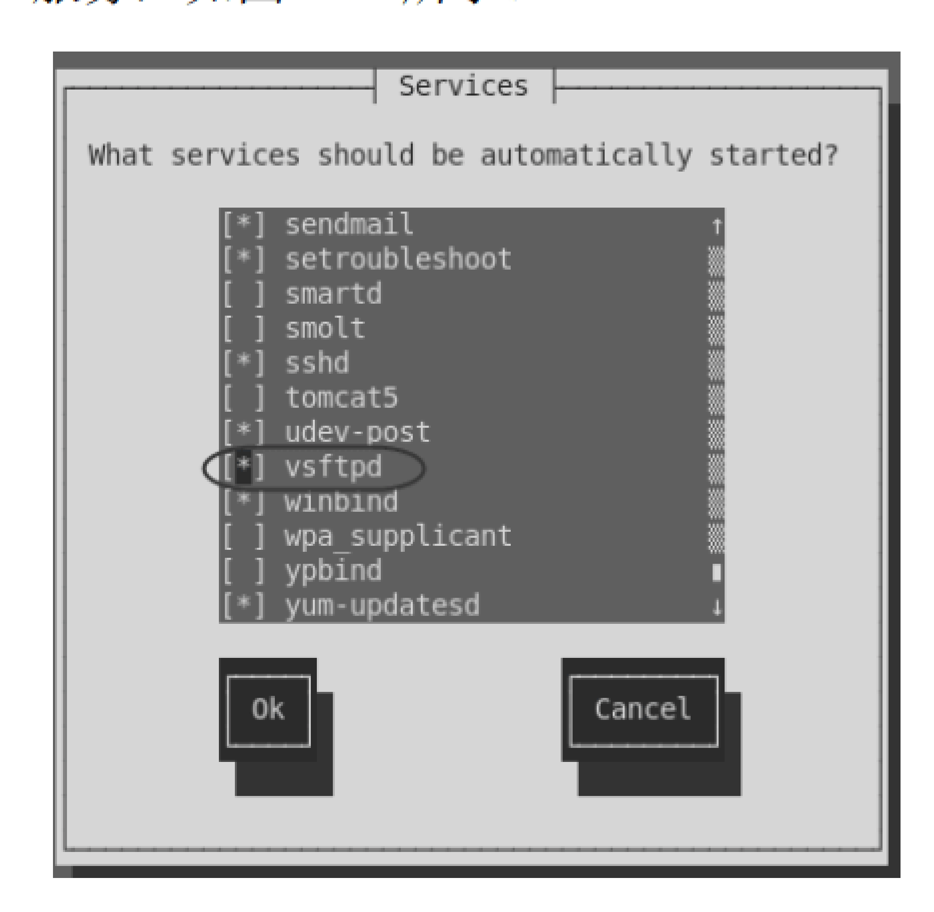



图 2.31　配置 vsftpd 服务

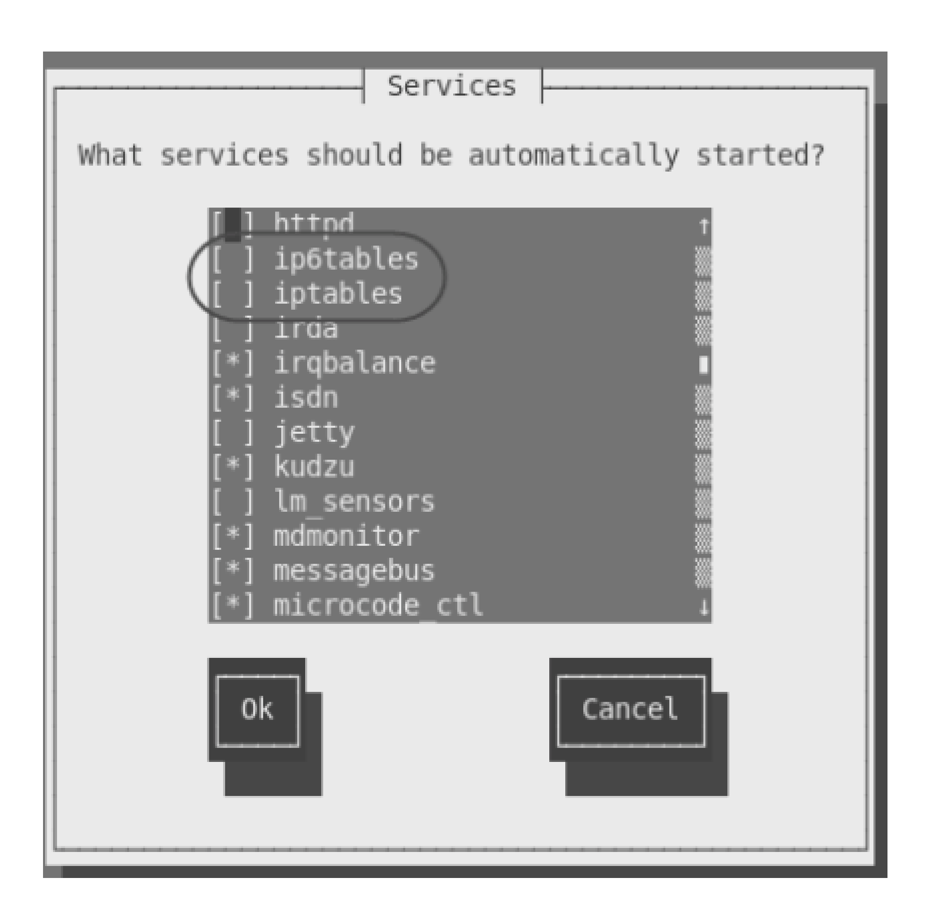



图 2.32　关闭防火墙

（3）服务器端用户配置。将目录/etc/vsftpd 下的文件 ftpusers 和 user_list 中的 root 用户注释掉，因为默认情况下，这两个文件中列出的用户是禁止访问 vsftpd 服务端的。

（4）SELINUX 的设置。安装系统时默认为强制选择 SELINUX，则在/etc/selinux/config

中 SELINUX 被设置成了 enforcing，应将其改为 disabled。

```
SELINUX=disabled
```

（5）客户端安装 flashfxp 终端软件。运行 flashfxp 后，进入“站点管理器配置”对话框。给被访问站点取个名字，将 IP 地址设置为虚拟机的 IP 地址，用户名设置为 root，密码设置为虚拟机的密码，然后设置远端和本地路径，最后单击“应用”|“连接”按钮访问服务器，配置如图 2.33 所示。

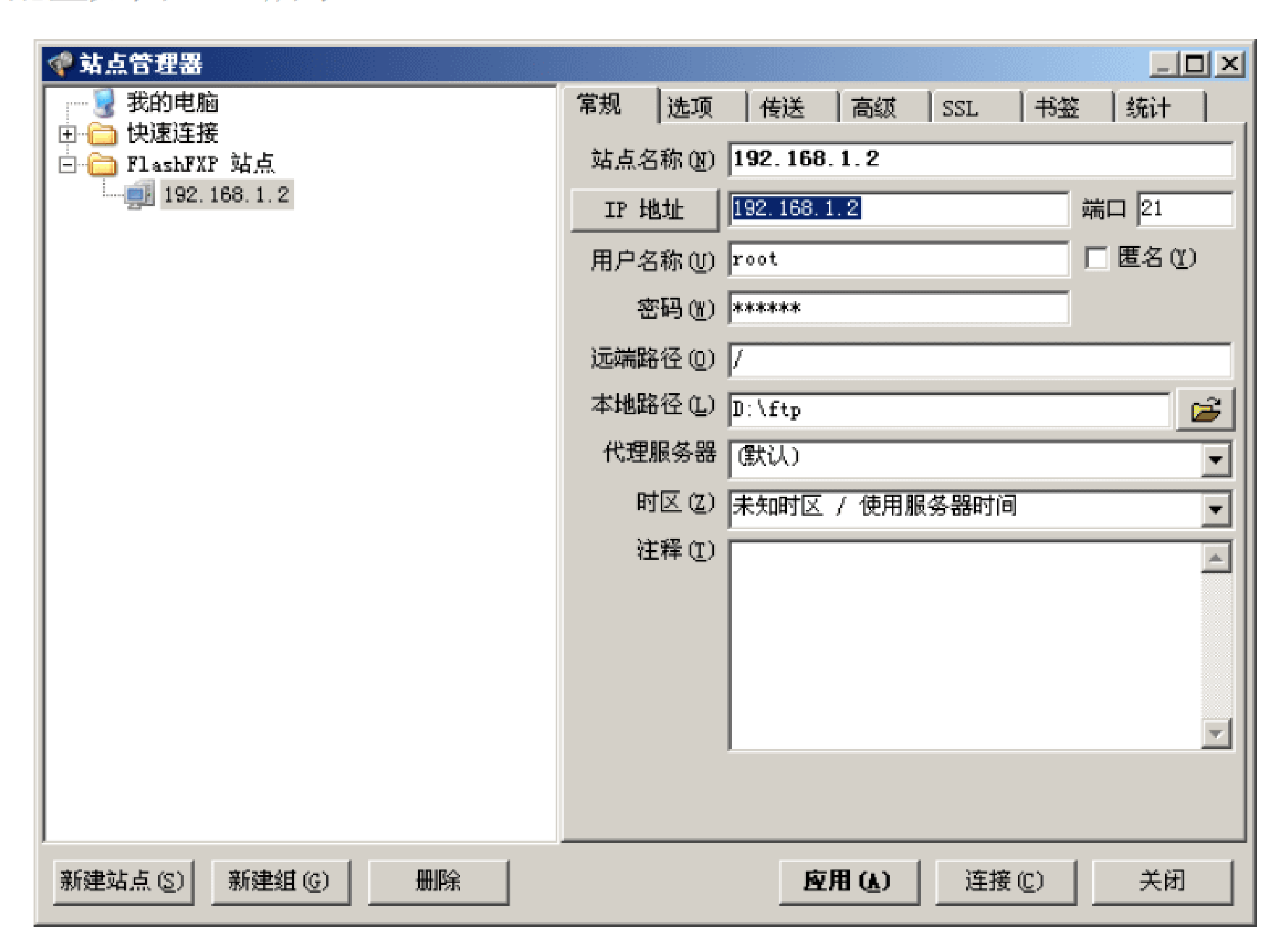



图 2.33　flashfxp 站点管理配置

# 2.2　交叉编译工具

交叉编译工具是为了使在上位机中编译的文件能够在不同平台的目标机中执行。本书介绍的目标平台均为 ARM 平台。

## 2.2.1　交叉编译工具安装

交叉编译工具的安装主要包括：编译 GNU binutils、获得 Linux 内核头文件、安装 Glibc 头文件、安装 GCC 第一阶段、安装 Glibc 和完全安装 GCC。一次成功安装的过程需要长达数小时的时间，因此在没有成功编译此工具经验时，可以参考一些稳定交叉编译器的版本。下面将详细介绍安装的过程，并提供相关的命令和解释供编译时参考。

（1）建立存放工具、源码目录和设置相关的环境变量。

```
#mkdir /usr/local/arm                       //建立工作目录
#cd /usr/local/arm                          //进入工作目录
```

```
#mkdir -p /usr/local/arm/src                    //建立安装文件的目录
#mkdir -p /usr/local/arm/sysroot                //建立系统根目录
#mkdir -p /usr/local/arm/bin                    //建立生成工具存放的目录
#mkdir -p /usr/local/arm/build                  //建立编译目录
#export HOST=i686-pc-linux-gnu                  //设置 HOST 环境变量
#export TARGET=arm-linux                        //设置 TARGET 环境变量
#export PREFIX=/usr/local/arm/                  //设置 PREFIX 环境变量
#export SYSROOT=/usr/local/arm/sysroot          //设置 SYSROOT 环境变量
#export PATH=$PATH:${PREFIX}/bin                //添加生成工具目录到 PATH 中
```

注意：在定义好环境变量后应该使用 echo 命令查看环境变量是否与预计的变量相符合。在重新打开 shell 时，或者在第二次接着编译交叉环境时都应该检查环境变量。例如当 PATH 没有设置正确时，在编译 glibc 时，就会因 arm-liunx-gcc 不存在而导致编译失败。

（2）从网站下载安装的源码包，需要下载的源码包如表 2.1 所示。

**表 2.1　源码包列表**

| 软件包名称 | 下 载 地 址 |
|---|---|
| binutils-2.16.tar.gz | http://ftp.gnu.org/gnu/binutils/ |
| linux-2.6.10.tar.gz | http://www.kernel.org/pub/linux/kernel/v2.6/ |
| gcc-3.4.4.tar.bz2 | http://ftp.gnu.org/gnu/gcc/gcc-4.4.0/ |
| glibc-2.3.5.tar.gz | http://ftp.gnu.org/gnu/glibc/ |
| glibc-linuxthreads-2.3.5.tar.gz | http://ftp.gnu.org/gnu/glibc/ |
| ioperm.c.diff | http://gcc.gnu.org/ |
| flow.c.diff | http://frank.harvard.edu/～coldwell/toolchain/ioperm.c.diff |
| t-linux.diff | http://frank.harvard.edu/～coldwell/toolchain/t-linux.diff |

将获得的源码复制到/usr/local/arm/src 目录下。目录结构如下：

```
|-- arm
|    |-- bin            //工具目录
|    |-- build          //编译目录
|    |-- src            //存放源码
|    |—sysroot          //编译过程生成系统的根目录
```

（3）编译 GNU binutils。

```
#cd ${PREFIX}/src
#tar xvfz binutils-2.16.tar.gz
#mkdir -p /usr/local/arm/build/binutils-2.16
#cd /usr/local/arm/build/binutils-2.16
#../../src/binutils-2.16/configure --prefix=${PREFIX} --target=${TARGET}
--with-sysroot=${SYSROOT} 2>&1 | tee configure.out
#make 2>&1 | tee make.out
#make install 2>&1 | tee -a make.out
```

编译选项注释：

```
--target=${TARGET}
```

这个选项是跟--host 一起表示编译生成的可执行文件运行在 HOST 上面，但这些可执

行文件服务的对象是 TARGET，也就是说用这些可执行文件连接和汇编出来的程序运行在 TARGET 上面。这里，默认就会使用主机的 GCC 编译器，因此省略了--host 选项。

```
--prefix=${RESULT_DIR}
```

告诉配置脚本当运行 make install 时把编译好的东西安装在 RESULT_DIR 目录下。

```
--with-sysroot=${SYSROOT}
```

SYSROOT 为系统根目录，生成相应的库放在 SYSROOT/lib 目录下，可执行文件放在 SYSROOT/sbin 下，配置文件放在 SYSROOT/etc 下，用户文件放在 SYSROOT/usr 下。

在上面的编译过程中如果出现问题，最好的方法就是删除编译目录下的所有文件，删除 Binutils 目录，重新解压 Binutils，再重新开始安装。安装成功后就会在/usr/local/arm/bin 目录下生成下面的工具：

```
arm-linux-addr2line arm-linux-c++filt arm-linux-nm      arm-linux-ranlib
arm-linux-strings
arm-linux-ar        arm-linux-gprof   arm-linux-objcopy arm-linux-readelf
arm-linux-strip
arm-linux-as        arm-linux-ld      arm-linux-objdump arm-linux-size
```

Binutils 是 GNU 工具之一，包括连接器、汇编器和其他用于目标文件和档案的工具，它是二进制代码的处理维护工具。安装 Binutils 工具包含的程序有 addr2line、ar、as、c++filt、gprof、ld、nm、objcopy、objdump、ranlib、readelf、size、strings、strip、libiberty、libbfd 和 libopcodes。对这些程序的简单解释如下。

- addr2line：把程序地址转换为文件名和行号。在命令行中给它一个地址和一个可执行文件名，它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件及行号。
- ar：用来建立、修改和提取归档文件。归档文件是包括了其他多个文件的一个较大的文件，从该文件中可以恢复其他文件内容。
- as：主要用来编译 GNU C 编译器 gcc 输出的汇编文件，编译产生的目标文件再由连接器 ld 进行连接操作。
- c++filt：连接器使用它来过滤 C++ 和 Java 符号，防止发生重载函数冲突。
- gprof：显示程序调用段的各种数据。
- ld：是连接器，它把所有编译产生的目标文件和归档文件结合在一起，重新定位数据，并连接符号引用。一般建立一个新编译程序的最后一步就是调用 ld。
- nm：列出目标文件中的符号。
- objcopy：把一种目标文件中的内容复制到另一种类型的目标文件中。
- objdump：显示一个或者更多目标文件的信息。使用选项来控制其显示的信息，它所显示的信息通常只有编写编译工具的人才感兴趣。
- ranlib：产生归档文件索引，并将其保存到这个归档文件中。在索引中列出了归档文件各成员所定义的可重分配目标文件。
- readelf：显示 elf 格式可执行文件的信息。
- size：列出目标文件每一段的大小及总体的大小。默认方式下，目标文件或者归档文件中的单个模块只产生一行输出。
- strings：打印某个文件的可打印字符串，这些字符串最少 4 个字符长，也可以使用

选项-n 设置字符串的最小长度。默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其他类型的文件它打印整个文件的可打印字符。这个程序对于了解非文本文件的内容很有帮助。

- strip：丢弃目标文件中的全部或者特定符号。
- libiberty：包含许多 GNU 程序都会用到的函数，这些程序有 getopt、obstack、strerror、strtol 和 strtoul。
- libbfd：二进制文件描述库。
- libopcode：用来处理 opcodes 的库，在生成一些应用程序的时候也会用到它。

（4）获得 Linux 内核头文件。并将头文件安装在${SYSROOT}/usr/include 目录下。

```
#cd ${PREFIX}/src
#tar xvfz linux-2.6.10.tar.gz
#cd linux-2.6.10
#make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
#make include/linux/version.h
#mkdir -p ${SYSROOT}/usr/include
#cp -a ${PREFIX}/src/linux-2.6.10/include/linux ${SYSROOT}/usr/include/
linux
#cp -a ${PREFIX}/src/linux-2.6.10/include/asm-arm ${SYSROOT}/usr/include/
asm
#cp -a ${PREFIX}/src/linux-2.6.10/include/asm-generic ${SYSROOT}/usr/
include/asm-generic
```

注意：执行 make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig 时，主要是为了选择一个对应 CPU 的类型。选择 System Type → ARM system type (Samsung S3C2410)→(X) Samsung S3C2410，如图 2.34 所示。

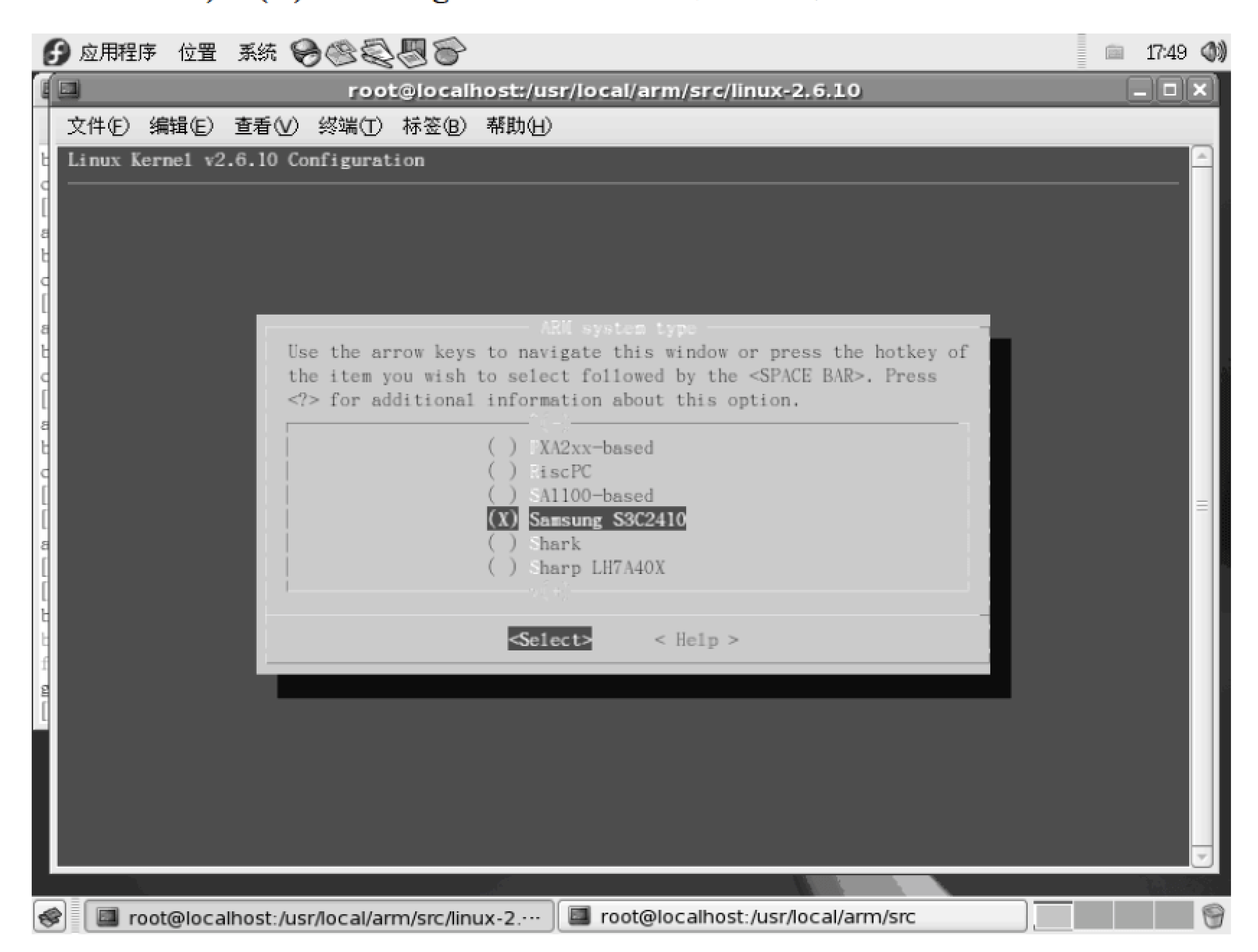



图 2.34　选择系统芯片类型

（5）安装 Glibc 头文件。Glibc 是 GUN C 库，它是编译 Linux 系统程序的重要组成部分。在安装 GNU C 库前需要先安装其头文件。

```
#cd ${PREFIX}/src
#tar xvfz glibc-2.3.5.tar.gz
#patch -d glibc-2.3.5 -p1 <ioperm.c.diff
#cd glibc-2.3.5
#tar xvfz ../glibc-linuxthreads-2.3.5.tar.gz
#cd ..
#mkdir -p /usr/local/arm/build/glibc-2.3.5-headers
#cd /usr/local/arm/build/glibc-2.3.5-headers
#../../src/glibc-2.3.5/configure --prefix=/usr --host=${TARGET} --enable-
add-ons=linuxthreads  --with-headers=${SYSROOT}/usr/include  2>&1  |  tee
configure.out
#make cross-compiling=yes install root=${SYSROOT} install-headers 2>&1 |
tee make.out
#touch ${SYSROOT}/usr/include/gnu/stubs.h
#touch ${SYSROOT}/usr/include/bits/stdio_lim.h
```

（6）安装 GCC 第一阶段。安装 GCC 共分为两个阶段，第一阶段是为了安装 ARM 交叉编译工具没有支持 libc 库的头文件。

```
#cd ${PREFIX}/src
#bunzip2 -c gcc-3.4.4.tar.bz2 | tar xvf -
#patch -d gcc-3.4.4 -p1 < flow.c.diff
                              //给 gcc-3.4.4 安装补丁文件 flow.c.diff
#patch -d gcc-3.4.4 -p1 < t-linux.diff
#mkdir -p /usr/local/arm/build/gcc-3.4.4-stage1
#cd /usr/local/arm/build/gcc-3.4.4-stage1
#../../src/gcc-3.4.4/configure --prefix=${PREFIX} --target=${TARGET}
--enable-languages=c --with-sysroot=${SYSROOT} 2>&1 | tee configure.out
#make 2>&1 | tee make.out
#make install 2>&1 | tee -a make.out
```

（7）安装 GNU C 库。安装了 Glibc 后才能对 GCC 进行完全安装。

```
#cd ${PREFIX}/src
#mkdir -p /usr/local/arm/build/glibc-2.3.5
#cd /usr/local/arm/build/glibc-2.3.5
#BUILD_CC=gcc CC=${TARGET}-gcc AR=${TARGET}-ar RANLIB=${TARGET}-ranlib
AS=${TARGET}-as LD=${TARGET}-ld ../../src/glibc-2.3.5/configure --prefix=
/usr
--build=i386-redhat-linux --host=arm-unknown-linux-gnu --target=arm-
unknown-linux-gnu
--without-__thread --enable-add-ons=linuxthreads --with-headers=
${SYSROOT}/usr/include
2>&1 | tee configure.out
#make 2>&1 | tee make.out
#BUILD_CC=gcc CC=${TARGET}-gcc AR=${TARGET}-ar RANLIB=${TARGET}-ranlib
AS=${TARGET}-as LD=${TARGET}-ld../../src/glibc-2.3.5/configure --prefix=
/usr build=${HOST} --host=${TARGET} -target=${TARGET} --without-__thread
--enable-add-ons=linuxthreads --with-headers=${SYSROOT}/usr/include 2>&1
| tee make.out
```

（8）完全安装 GCC。

```
#cd ${PREFIX}/src
#mkdir -p /usr/local/arm/build/gcc-3.4.4
#cd /usr/local/arm/build/gcc-3.4.4
#../../src/gcc-3.4.4/configure --prefix=${PREFIX} --target=${TARGET}
```

```
--enable-languages=c --with-sysroot=${SYSROOT} 2>&1 | tee configure.out
#make 2>&1 | tee make.out
#make install 2>&1 | tee -a make.out
```

编译完成后会在/usr/local/arm/bin 目录下增加交叉编译工具，具体生成的工具如下：

```
arm-linux-addr2line  arm-linux-cpp     arm-linux-gcov    arm-linux-objdump
arm-linux-strings
arm-linux-ar        arm-linux-gcc       arm-linux-ld     arm-linux-ranlib
arm-linux-strip
arm-linux-as       arm-linux-gcc-3.4.4  arm-linux-nm     arm-linux-readelf
arm-linux-c++filt   arm-linux-gccbug   arm-linux-objcopy  arm-linux-size
```

（9）删除源码目录、临时目录和一些中间目录，得到 arm-linux、bin、lib、libexec 和 share 目录。

注意：编译交叉编译器是一个很耗时的工作，对于实际项目的作用并不大。除非在某些应用程序或者驱动模块已经通过测试进入成品库，而这些应用程序或驱动模块依赖某个版本 GCC 或 glibc，同时修改和测试应用程序或驱动模块的工作量相对非常复杂，此时可以选择需要的版本进行建立交叉编译环境。一般情况下，建议读者直接使用开发板厂商提供的交叉编译器，或者在网上下载稳定的交叉编译器。目前针对 2.4 内核的稳定版本为 2.95.3，针对 2.6 内核的稳定版本为 3.4.1。笔者目前还使用开发商提供的 4.3.2 版本。在后面的内核移植和驱动移植过程中使用的就是 4.3.2 版本。

## 2.2.2　交叉编译器测试

在使用新建立的交叉编译器前需要对其进行简单的测试，查看其生成的文件是否可以移植到 ARM 平台的开发板上运行。

（1）对编译的交叉工具链进行简单的测试。将 arm-linux-gcc 添加到环境变量中。

```
# vi /etc/profile
```

找到# Path manipulation 部分，添加 arm-linux-gcc 所在目录。修改后保存配置重启系统配置生效。

```
# Path manipulation
if [ "$EUID" = "0" ]; then
      pathmunge /sbin
      pathmunge /usr/sbin
      pathmunge /usr/local/sbin
fi
```

修改如下：

```
# Path manipulation
if [ "$EUID" = "0" ]; then
      pathmunge /sbin
      pathmunge /usr/sbin
      pathmunge /usr/local/sbin
      pathmunge /usr/local/arm/bin
fi
```

（2）编写简单的测试程序，查看程序适应的体系结构。

```
#include <stdio.h>
int main()
{
        printf("test arm-linux-gcc");
        return 0;
}
```

对上面的程序进行交叉编译，并查看其头文件信息，看其运行属于哪种平台体系结构。

```
#arm-linux-gcc -o test test.c                //交叉编译 test.c，生成 test
# readelf test -h                            //使用 readelf 命令查看头文件信息
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            ARM
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x8378
  Start of program headers:          52 (bytes into file)
  Start of section headers:          3600 (bytes into file)
  Flags:                             0x2, has entry point, GNU EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
  Size of section headers:           40 (bytes)
  Number of section headers:         34
  Section header string table index: 31
```

可以看出该文件运行的环境为 ARM。

## 2.3　超级终端和 Minicom

在需要对目标板进行查看、操作或者目标板和上位机进行文件传输和通信时，需要安装终端软件。通过终端软件来对目标板进行配置，或者执行目标板上的程序与主机进行通信。

### 2.3.1　超级终端软件的安装

在 Windows 环境中，一般使用系统自带的超级终端软件，或者安装其他的终端软件。下面介绍超级终端软件的使用和设置。

执行“开始”|“所有程序”|“附件”|“通信”|“超级终端”命令，打开超级终端软件，如图 2.35 所示。可以任意为其取名字，可以以开发板型号作为名称。

开发板一般是通过串口线和 PC 连接。在选择连接使用的端口时，如果有计算机串行接口，则一般默认选择 COM1。这里使用的计算机没有串口，采用的是 USB 转串口方式与开发板进行连接。图 2.36 是对连接端口的选择。如果第一次设置不清楚时，可以查看设备管理器，如图 2.37 所示。

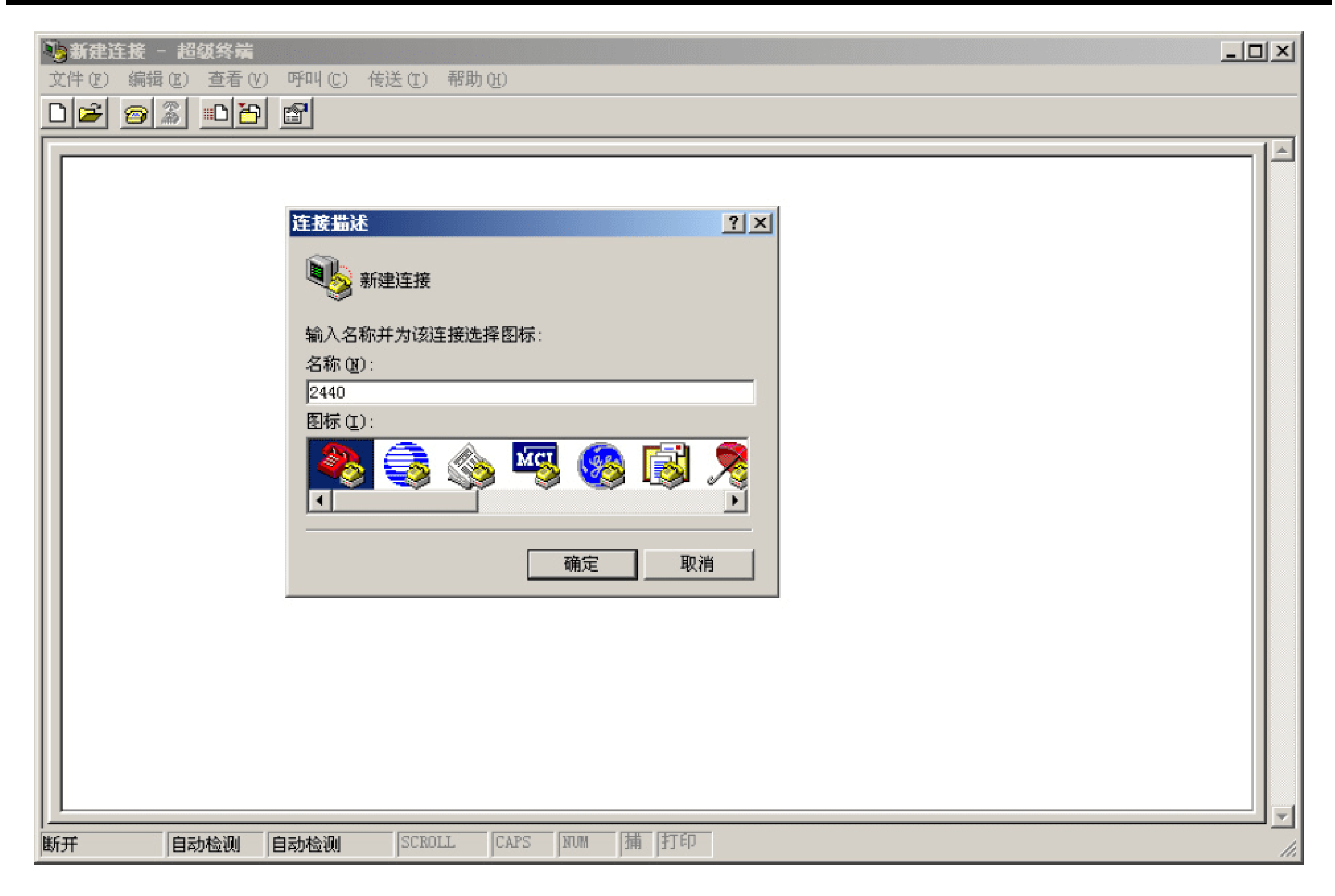



图 2.35　打开超级终端软件界面

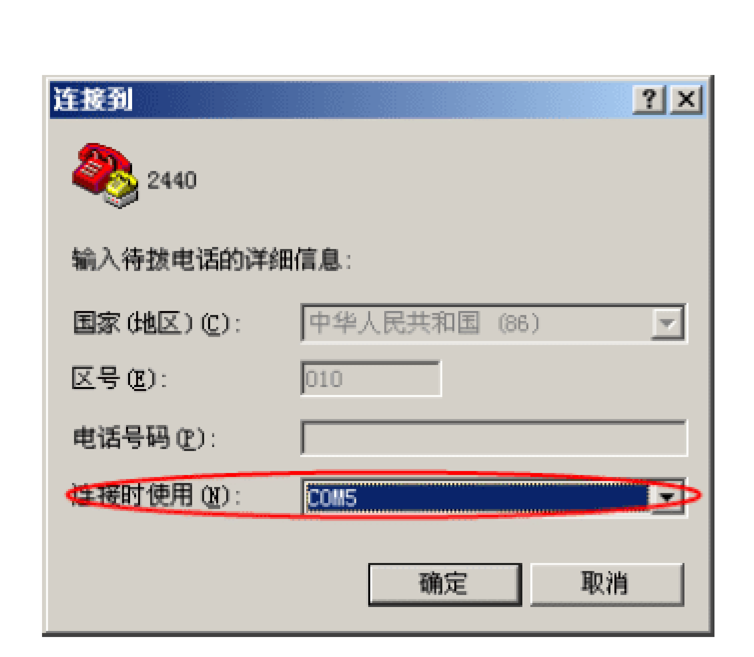



图 2.36　超级终端连接端口选择

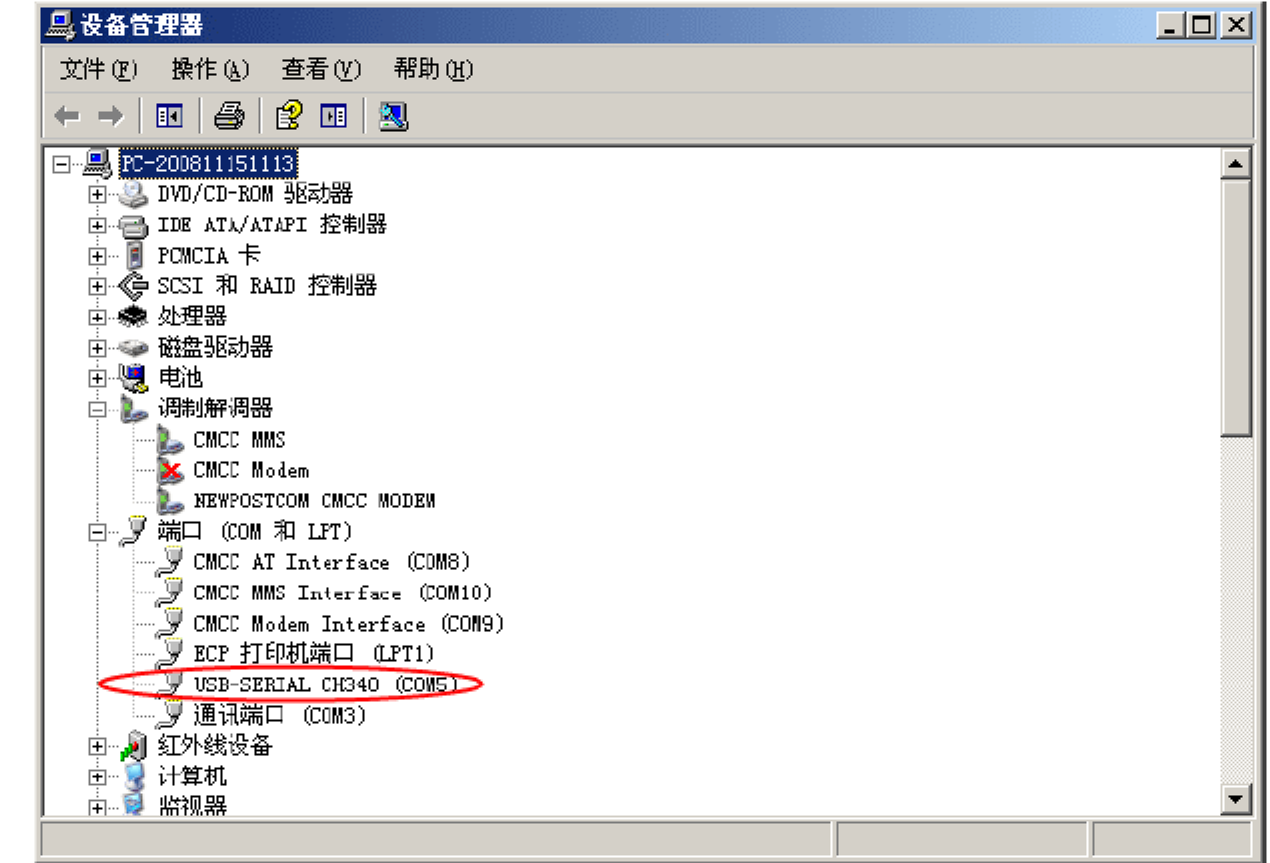



图 2.37　确定与开发板相连的端口

对端口参数的设置包括比特率、奇偶校验、数据流控制等，其相应的选择如图 2.38 所示。最后单击“确定”按钮保存配置，下次可以直接使用。

## 2.3.2　Minicom 使用

Minicom 是 Linux 系统中的终端软件。在 Linux 系统中可以通过此软件访问目标板。如果上位机中带有串口，那么 ttyS0 代表 COM1，ttyS1 代表的就是 COM2。如果在上位机

中不带串口，那么需要编译加载 usbserial 模块。加载模块后可以在 dev 下面生成 ttyUSB0。编译内核、驱动的方法和加载驱动的方法将在后面介绍。下面介绍 minicom 的设置。

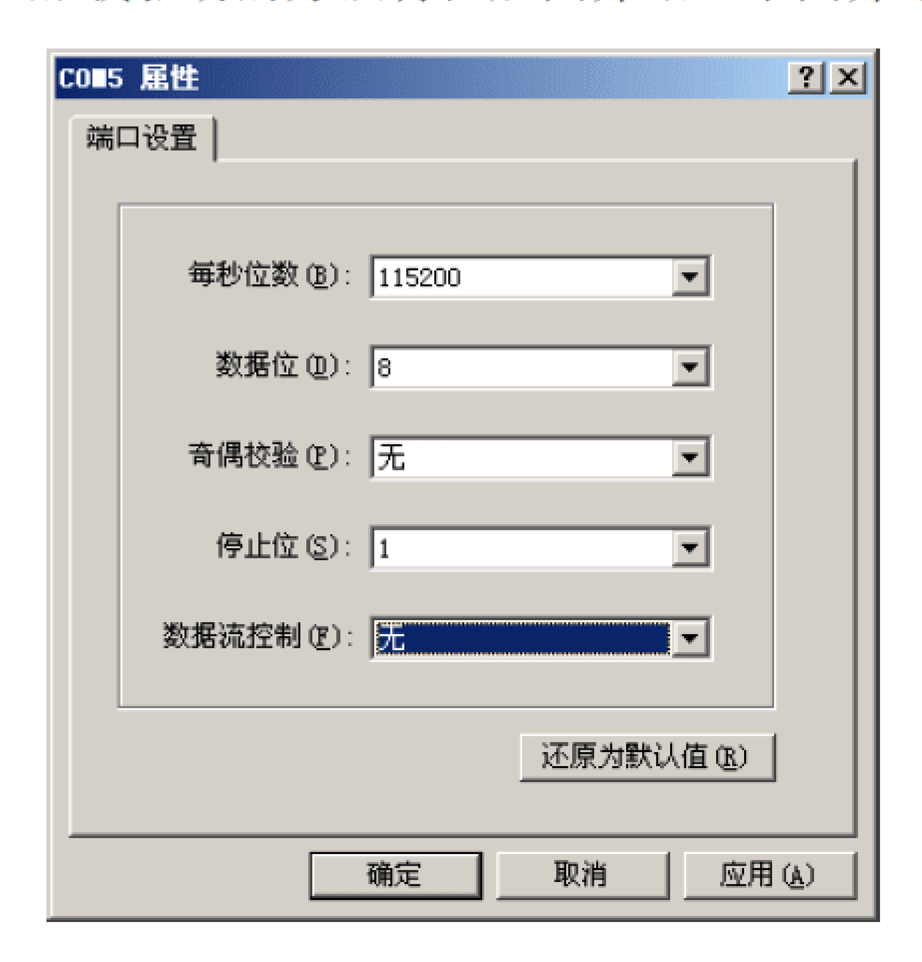



图 2.38　端口参数设置

（1）在 Shell 中使用 minicom –s 命令进行配置界面，如图 2.39 所示。

```
#minicom -s
```

```
应用程序  位置  系统
root@localhost:~
文件(F)  编辑(E)  查看(V)  终端(T)  标签(B)  帮助(H)

+-----[configuration]------+
| Filenames and paths      |
| File transfer protocols  |
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
+--------------------------+
```

图 2.39　Minicom 的配置界面

（2）使用键盘的上下键对光标进行操作，选择 Serial port setup 项后按下 Enter 键确定进入此项进行配置。如果选错选项可使用 Esc 键退出。进入串口的配置界面如图 2.40 所示。

（3）通过按下键盘的 A、B、C、D、E、F、G 键选择进入各个参数项的配置。配置完成后按下 Enter 键确认配置。对波特率、数据位和停止位参数配置，如图 2.41 所示，最终配置如图 2.42 所示。最后退出并保存配置。

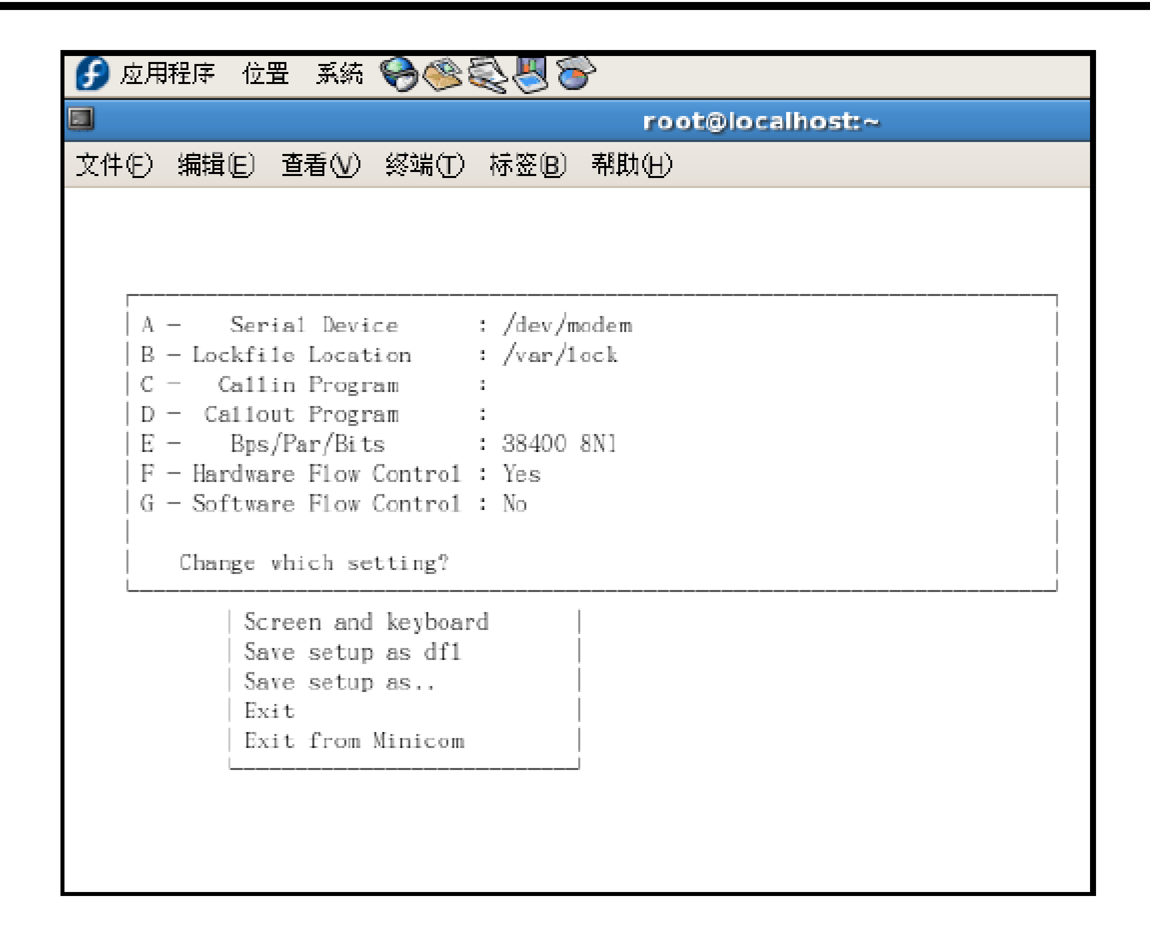



图 2.40　串口的配置界面

```
应用程序  位置  系统
root@localhost:~
文件(F)  编辑(E)  查看(V)  终端(T)  标签(B)  帮助(H)

              [Comm Parameters]
A -    Serial   | Current: 115200 8N1
B - Lockfile L  |
C -   Callin P  |  Speed          Parity        Data
D -  Callout P  |
E -    Bps/Par  | A: 300         L: None       S: 5
F - Hardware F  | B: 1200        M: Even       T: 6
G - Software F  | C: 2400        N: Odd        U: 7
                | D: 4800        O: Mark       V: 8
   Change whic  | E: 9600        P: Space
                | F: 19200                   Stopbits
     Screen     | G: 38400                     W: 1
     Save s     | H: 57600                     X: 2
     Save s     | I: 115200      Q: 8-N-1
     Exit       | J: 230400      R: 7-E-1
     Exit f     |
                | Choice, or <Enter> to exit?
```

图 2.41　波特率、数据位和停止位配置

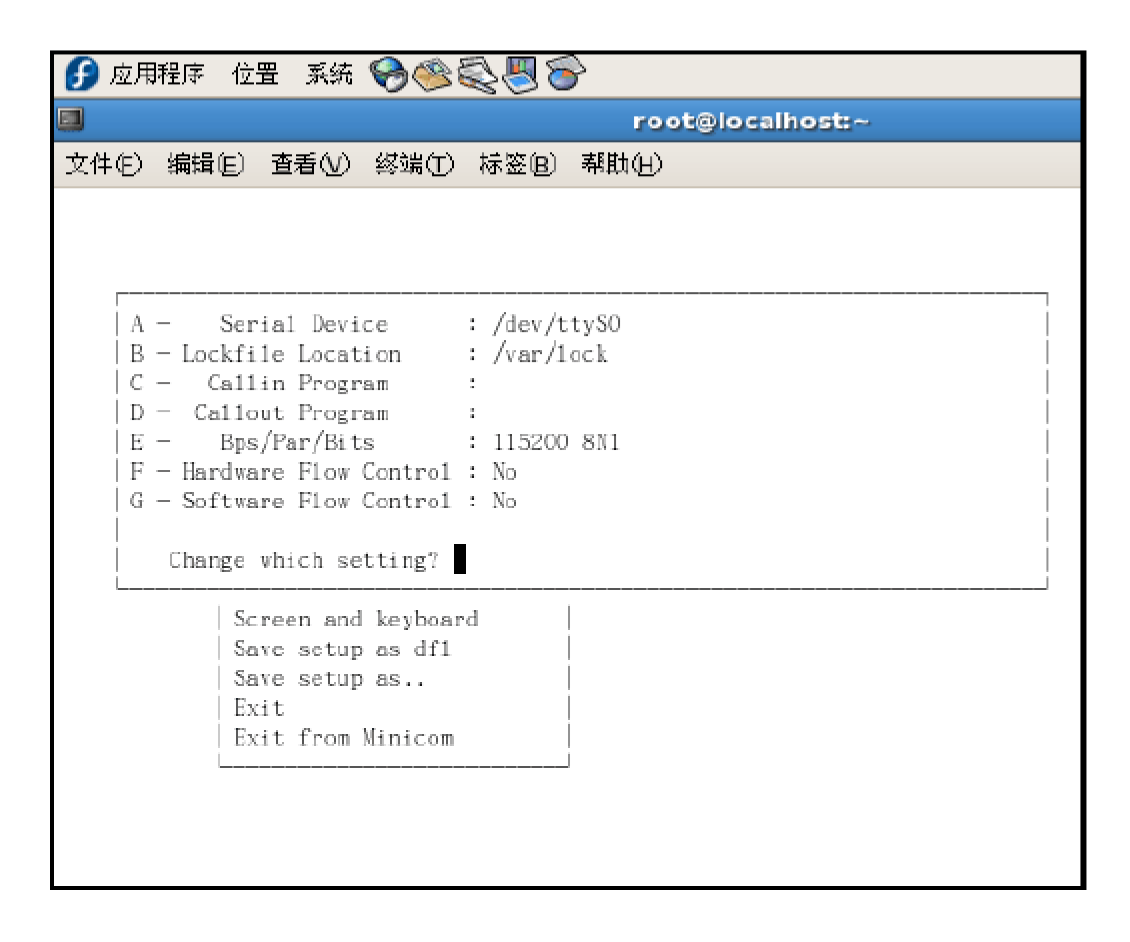



图 2.42　最终配置

### 2.3.3　SecureCRT 使用

SecureCRT 也是一款功能强大的终端软件，其使用环境为 Windows 环境。对于安装过程比较简单不予介绍。本节只介绍其第一次使用时的配置过程。配置一次后，以后直接连接即可。

（1）通过菜单 File | Connect 或 File | Quick Connect 命令均可进入配置界面，或者通过工具栏的 Connect 按钮和 Quick Connect 按钮进入配置界面。两者的配置界面不同但是参数是一样的。这里举例 Quick Connect 配置界面，如图 2.43 所示。

（2）按下 Connect 按钮保存配置并进入连接状态。下次运行此软件时可以直接选择此连接，如图 2.44 所示。直接选择 Connect 按钮进入连接状态。

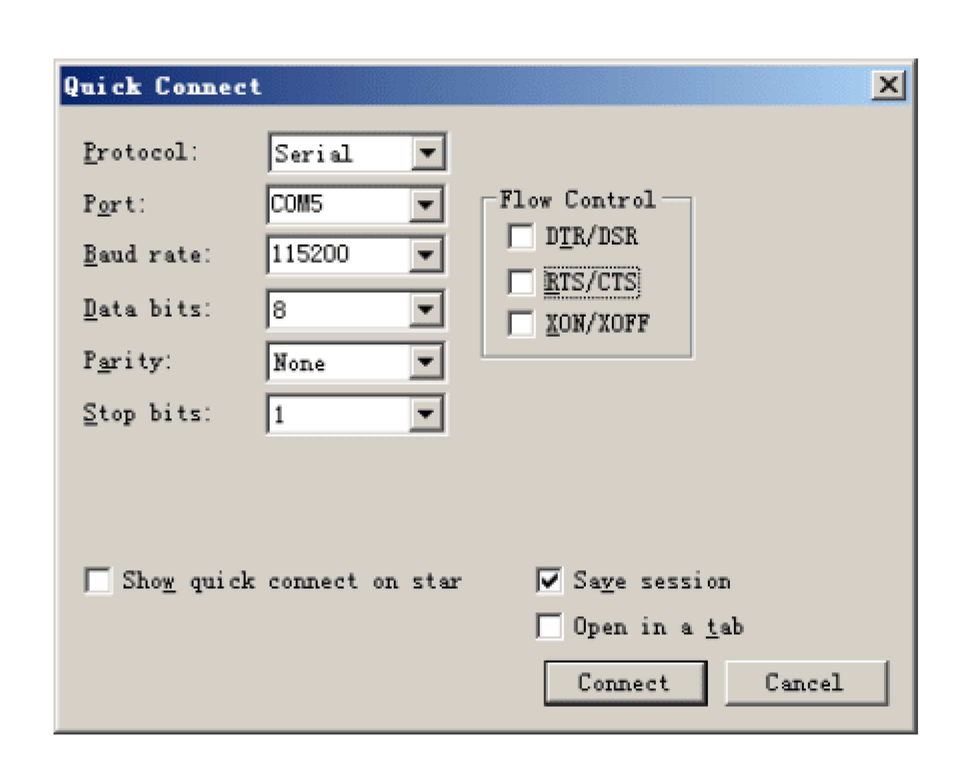




图 2.43　SecureCRT 串口配置

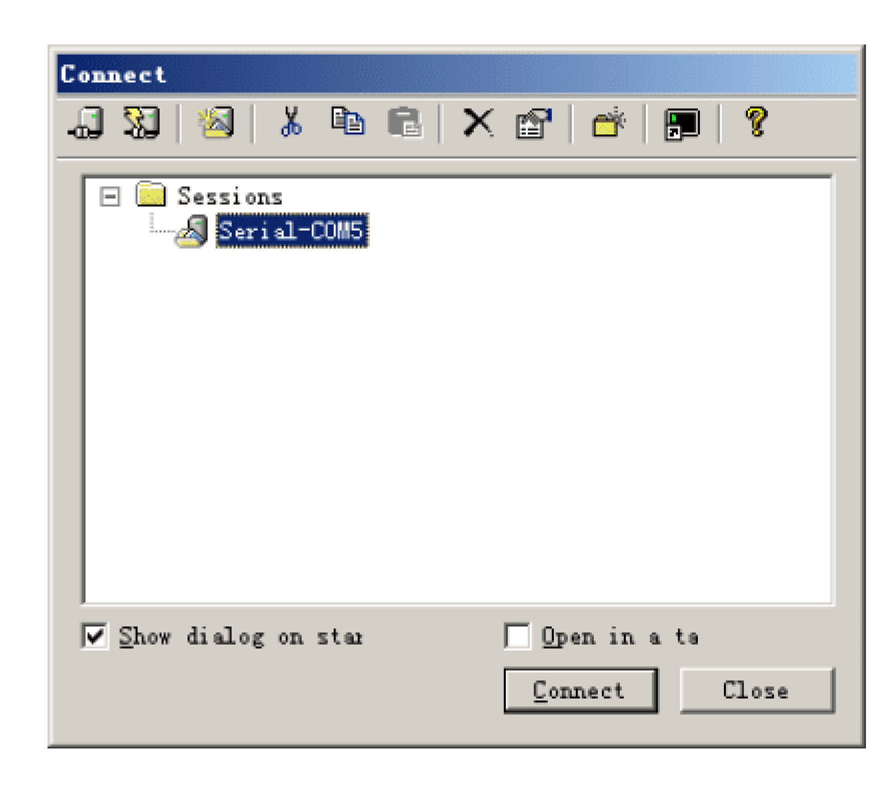



图 2.44　重新运行 SecureCRT 界面

注意：以上 3 个终端软件任意使用一个即可，并非都要进行安装。

## 2.4　内核、文件系统加载工具

内核、文件系统加载工具是嵌入式开发必备的工具，在购买开发板时，会得到配套的这类工具。不同的公司提供的工具和方法略有不同。这里针对重要的地方加以介绍。

### 2.4.1　烧写 Bootloader

可以使用超级终端的“传送”|“发送文件”命令进入发送文件对话框，使用 Xmodem 协议和 Kermit 协议发送 Bootloader 的各个文件。选择协议如图 2.45 所示。

JTAG 烧写 Bootloader 程序过程如下：

（1）确定 JTAG 烧写器与开发板的 JTAG 口相连，然后连接并口线和上位机。

（2）运行 JTAG Server 软件，打开开发板电源。对 JTAG 工具进行配置，选择 Setting | LPT Jtag Setting 命令进入配置界面，如图 2.46 所示。然后可以设置配置界面，如图 2.47 所示。

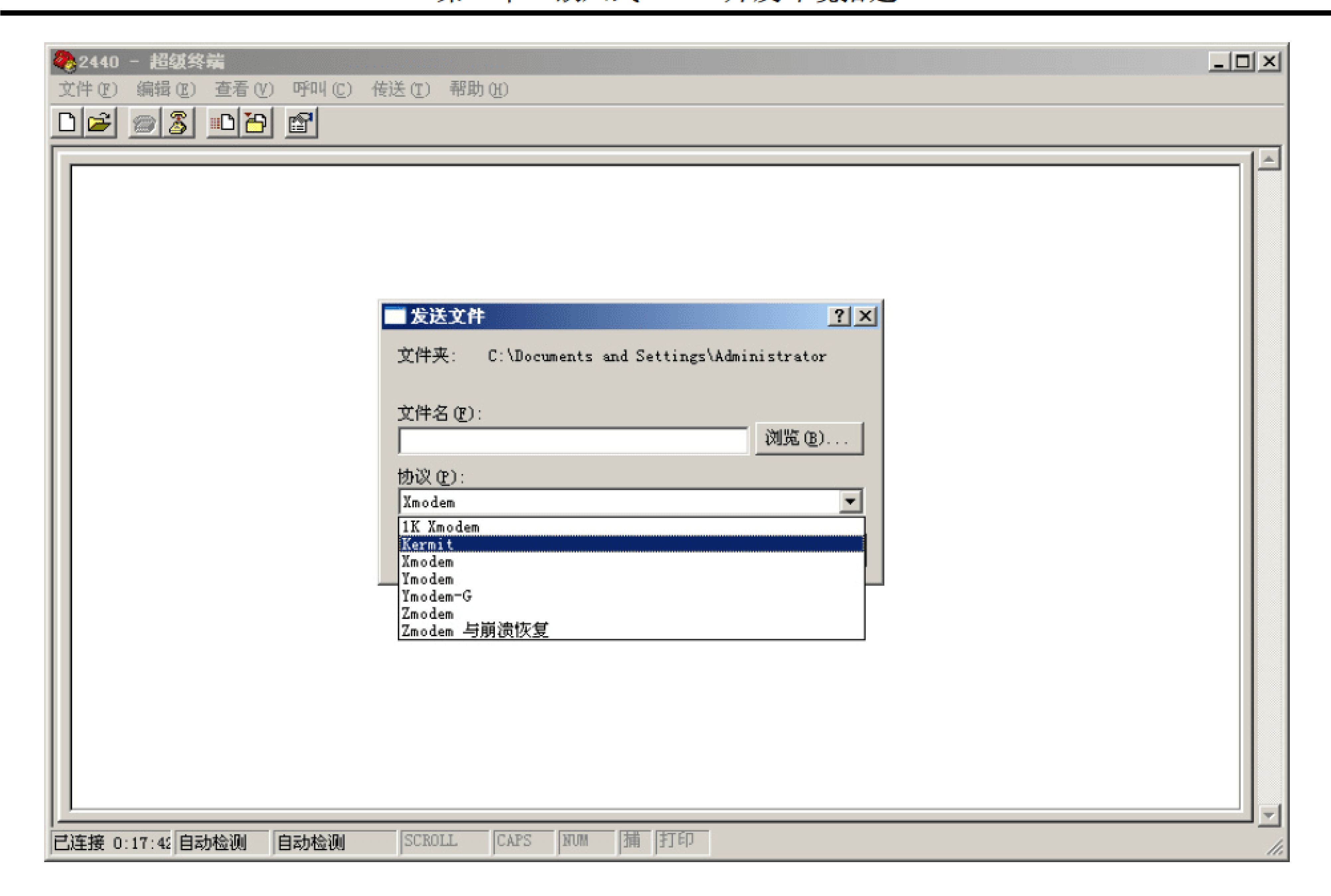



图 2.45　使用超级终端烧写 Bootloader

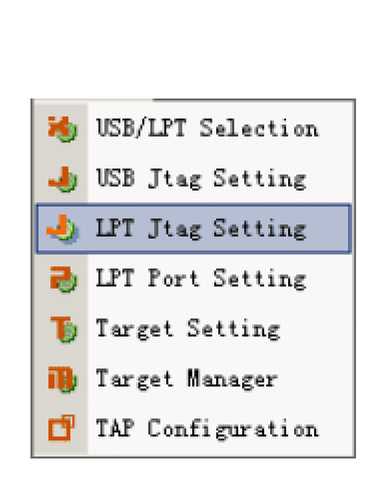



图 2.46　选择 JTAG 配置

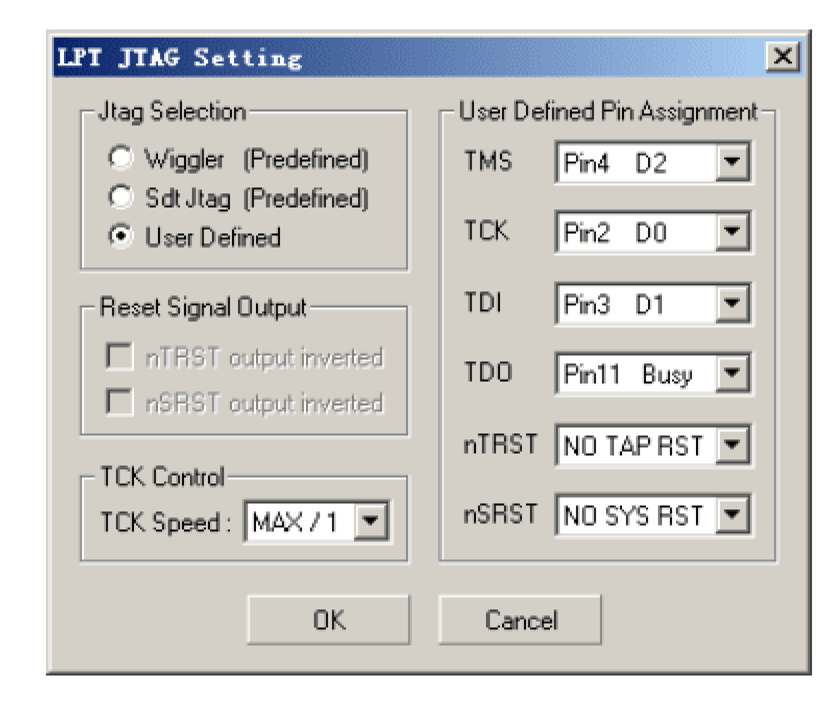



图 2.47　JTAG 配置界面

（3）正确配置后，可以识别开发板 CPU 类型，如图 2.48 所示。

（4）并口的设置。如果不能正确识别 CPU 类型，则应该在设备管理器中检验并口是否设置正确。在设备管理器中选择 ECP 打印机端口，如图 2.49 所示。

（5）进入打印机端口属性对话框后，选择“资源”标签，在其中查看打印机是否为 ECP 类型；起始地址是否为 0378，如图 2.50 所示。如果设置不正确则应该进入 BIOS 对并口设置进行修改。

（6）选择 Script | Init Script 命令，然后单击 Load 按钮加载.his 文件进行初始化，如图 2.51 所示。

（7）选择 Flasher | Start H-Flasher 命令进入烧写界面，加载对应 CPU 类型的.hfc 文件。选择烧写命令如图 2.52 所示，加载 hfc 文件后界面如图 2.53 所示。

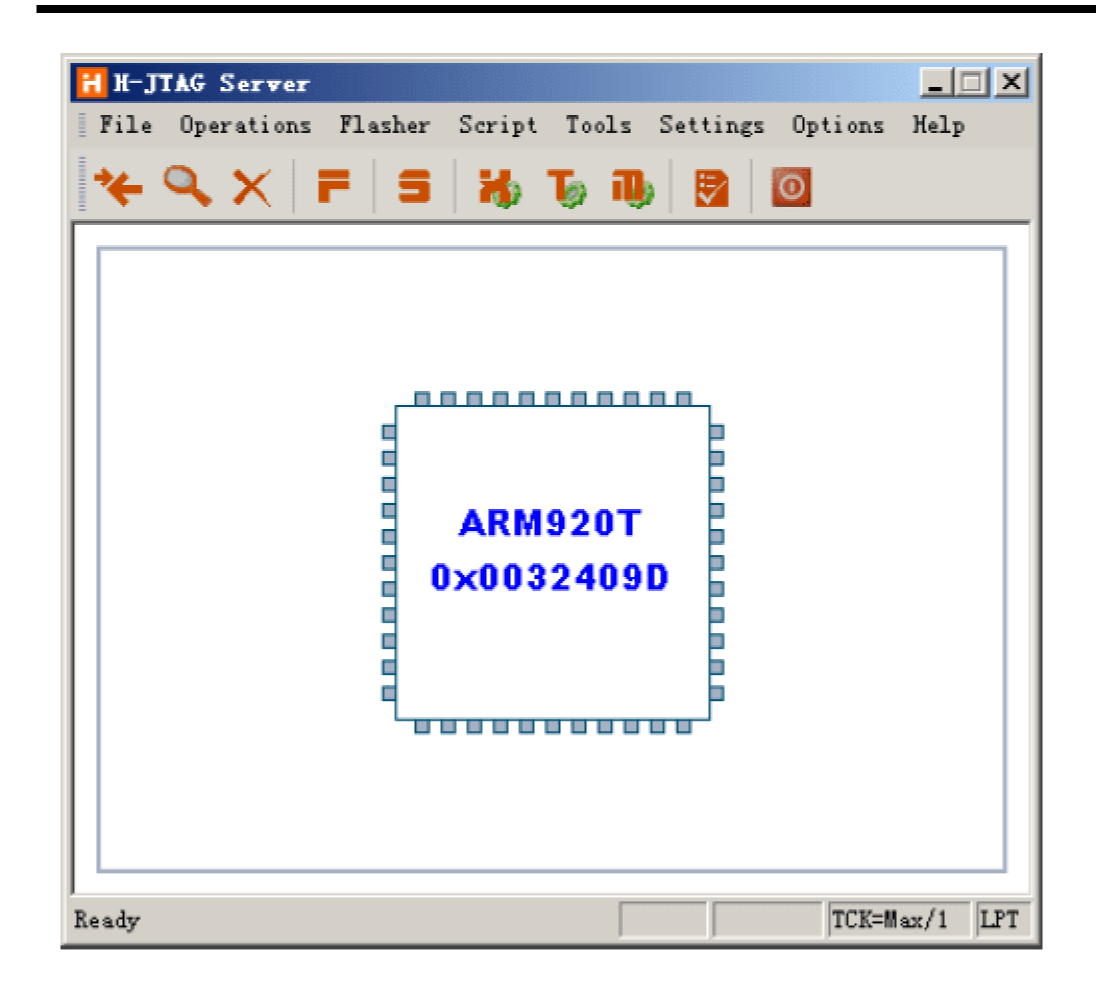



图 2.48　正确识别 CPU 类型




图 2.49　选择 ECP 打印机端口

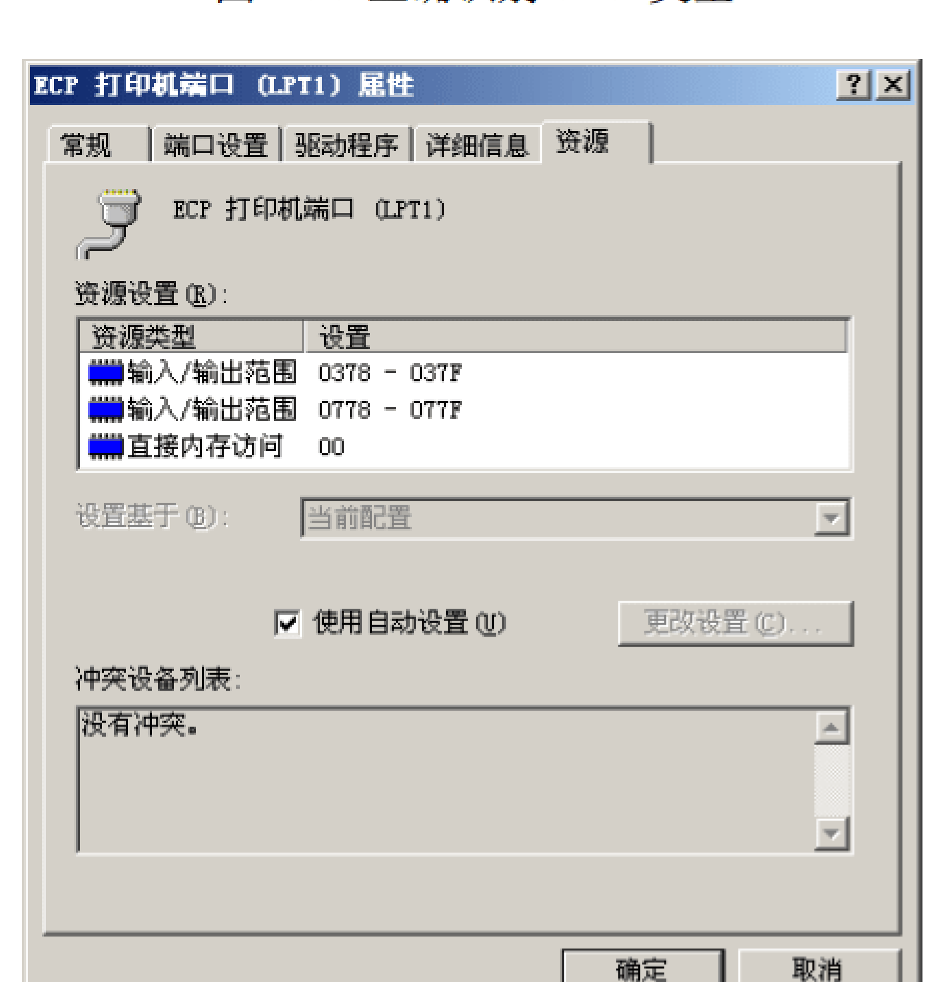



图 2.50　查看打印机端口属性

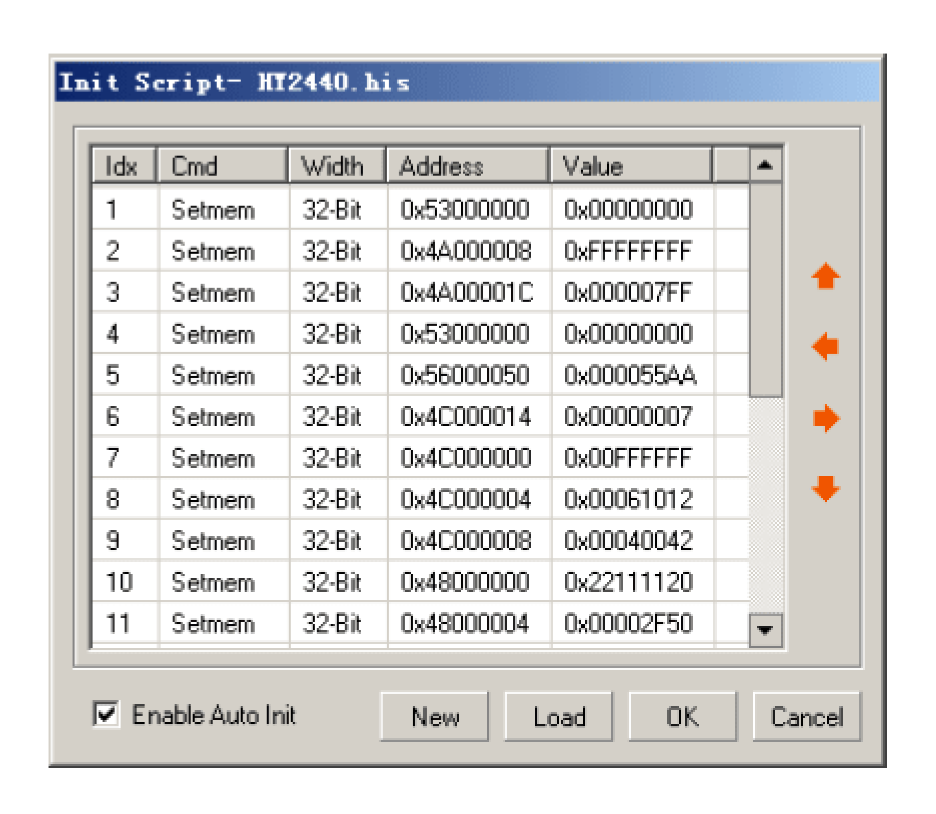



图 2.51　初始化 JTAG Server

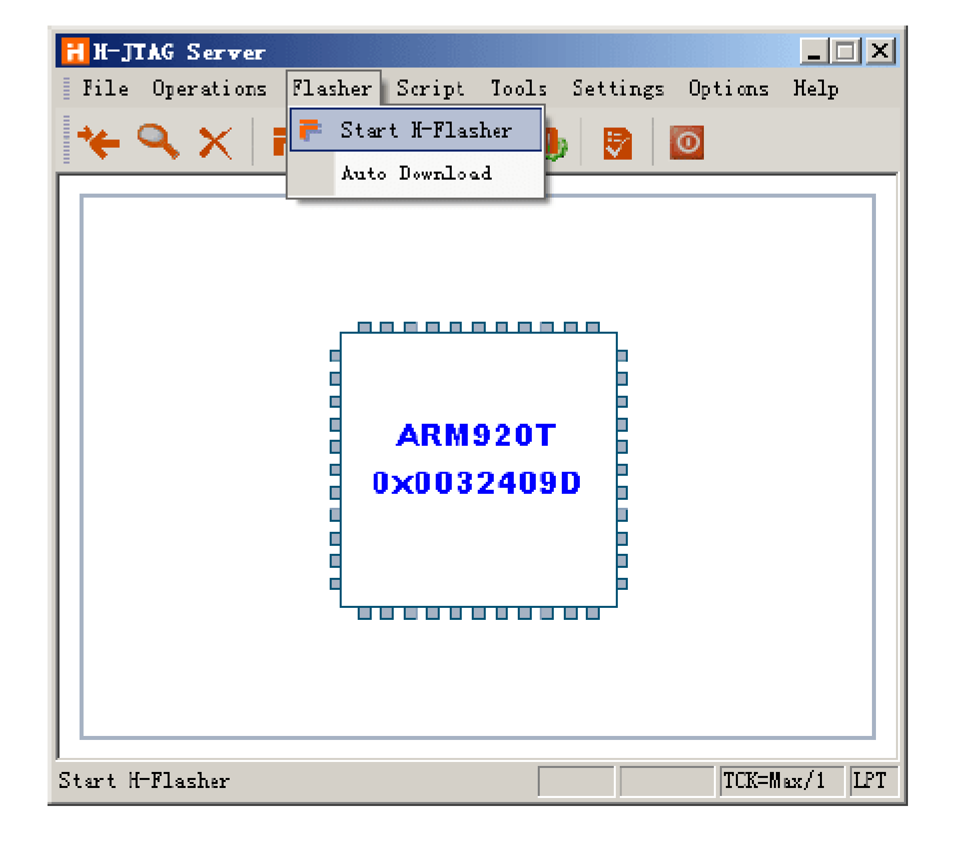



图 2.52　选择烧写命令

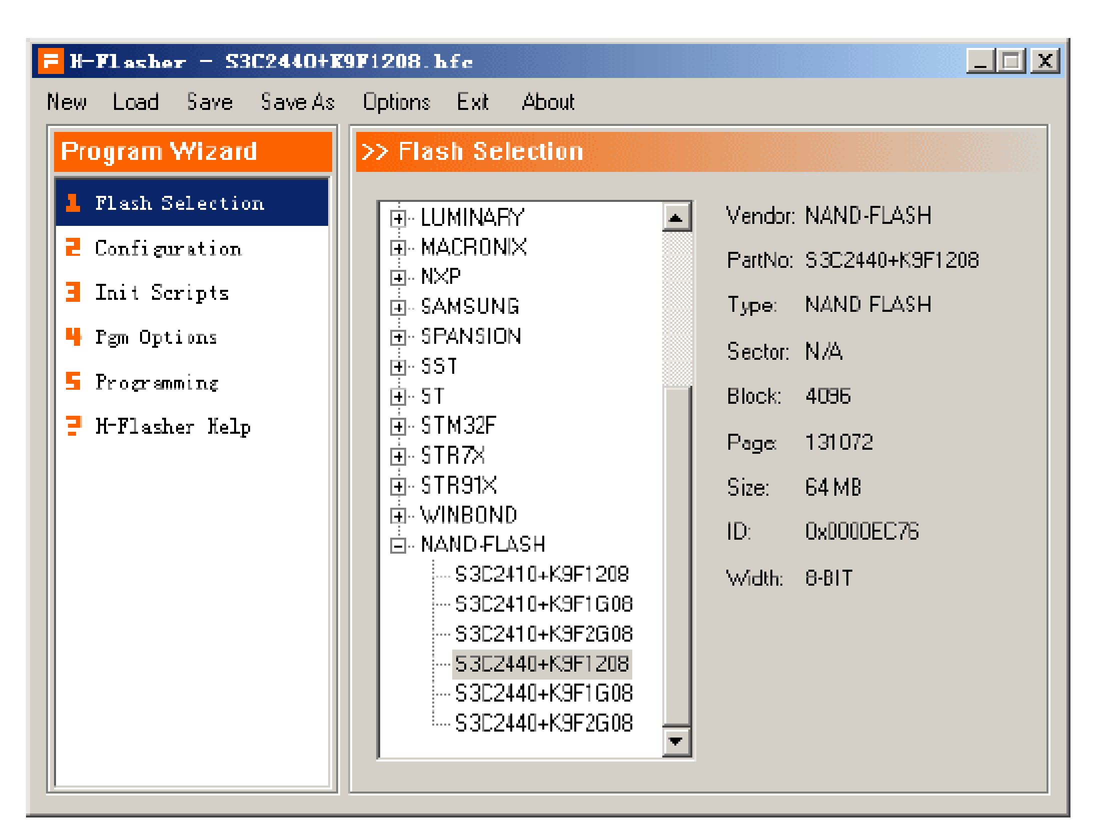



图 2.53　加载 hfc 文件

（8）在 Programming 项中指定烧写文件的路径，如图 2.54 所示。按下 Program 按钮就可以使用 JTAG 口烧写 Bootloader。

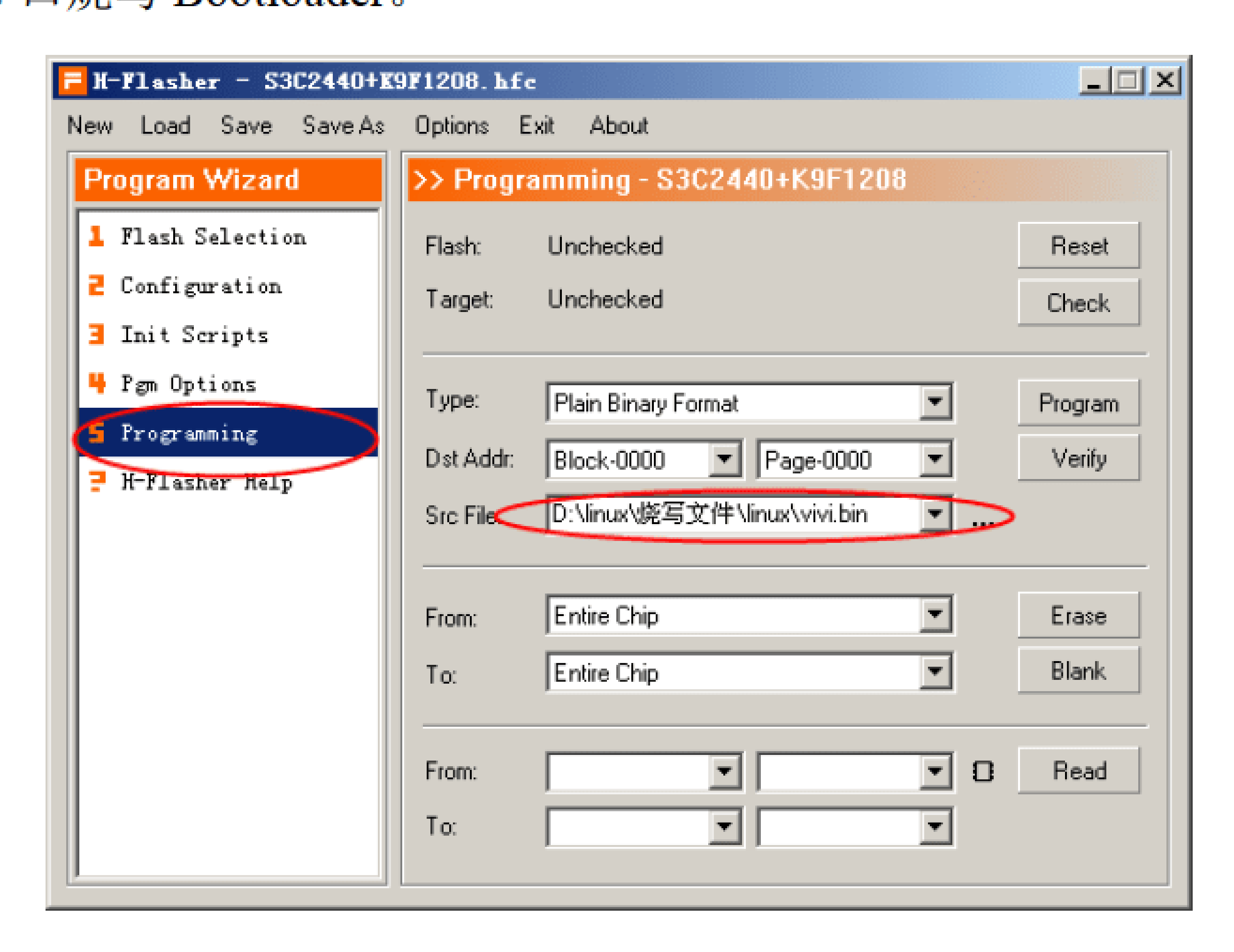



图 2.54　指定烧写文件的路径

（9）如果开发板中已经安装了系统，那么重启开发板，然后立刻单击 Program 按钮进行烧写。烧写过程如图 2.55 所示。

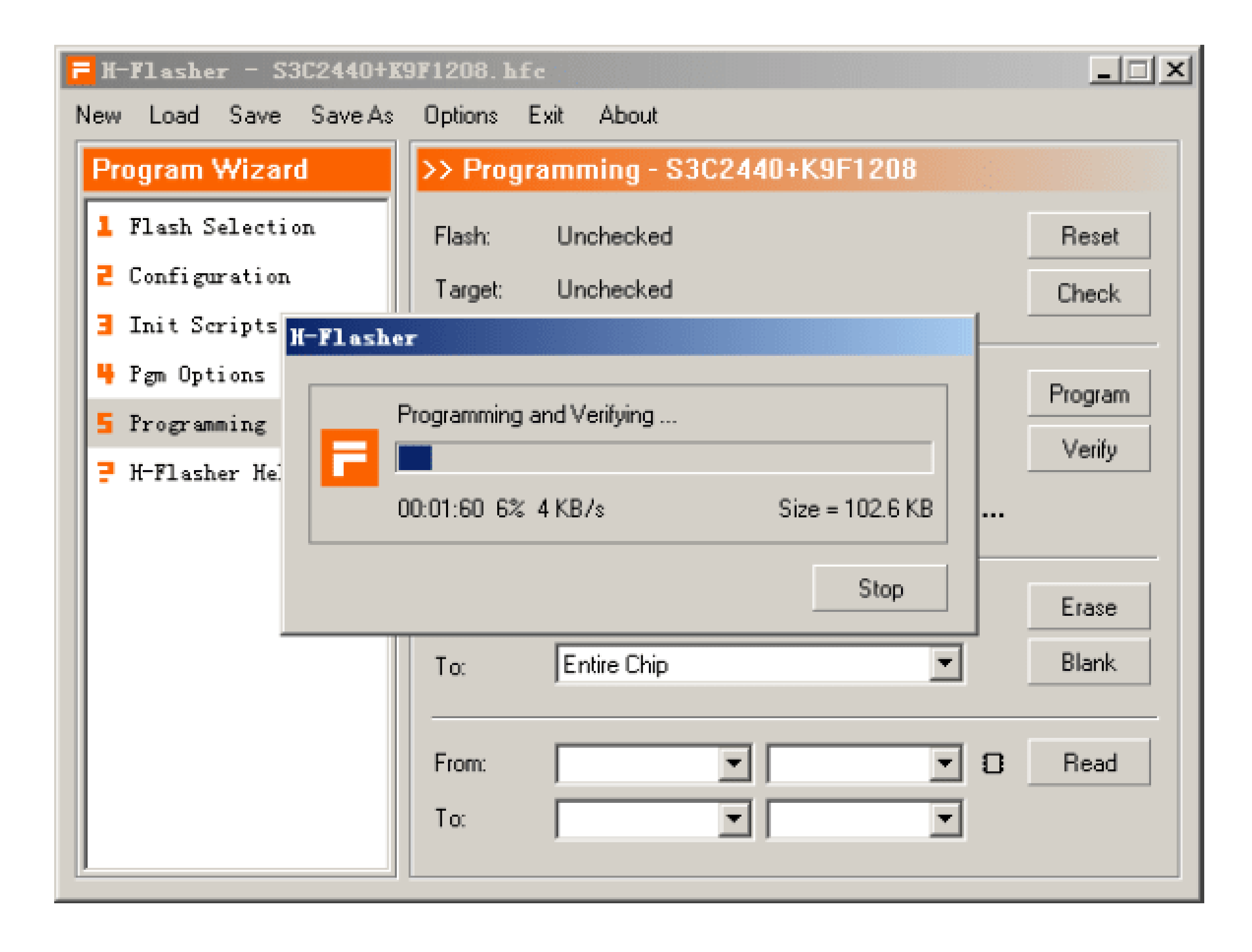



图 2.55　烧写 Bootloader 过程

（10）正确烧写完成后的提示界面如图 2.56 所示，至此完成整个 Bootloader 烧写过程。

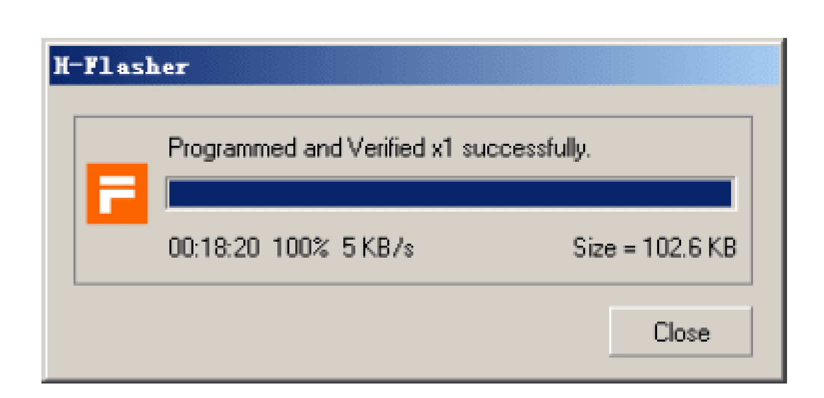



图 2.56　正确烧写完成

## 2.4.2　内核和文件系统下载

有些公司提供网口下载内核和文件系统的方式；而有些公司采用 USB 方式下载文件系统和内核。采用网口方式下载时需要安装 TFTP 工具，然后设置正确的 IP 地址和下载文件路径，同时需要在 U-boot 中设置服务器的 IP 为上位机的 IP 地址。设置开发板的 IP 地址与上位机的 IP 地址为同一个网段，在上位机中建立 TFTP 服务器后，通过终端软件输入 tftp 命令下载内核和文件系统。

使用 USB 端口下载内核和文件系统时，需要安装 DNW 工具。开发板需要通过串口与上位机相连。运行 DNW 工具，选择 Configuration | Option 命令进入配置界面，如图 2.57 所示。

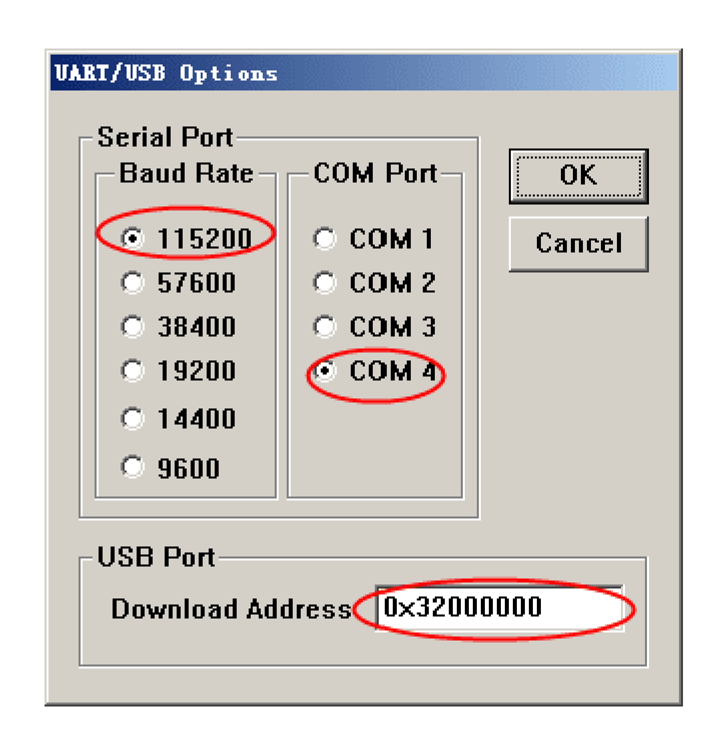



图 2.57　对串口通信的配置

选择 Serial | Connect 命令，建立与开发板的连接。选择建立连接命令如图 2.58 所示，成功连接后如图 2.59 所示。

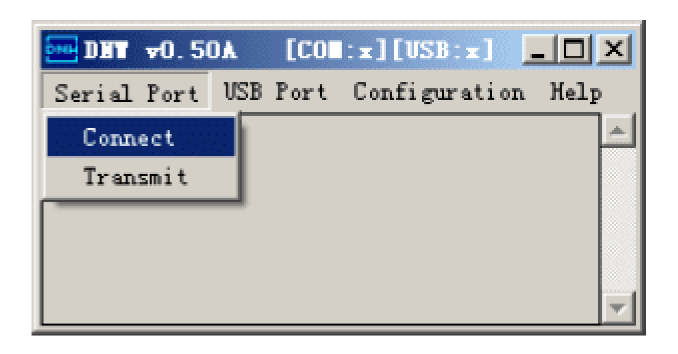



图 2.58　建立连接命令

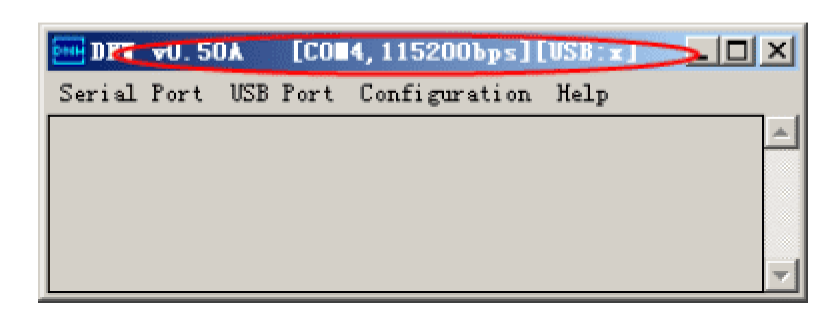



图 2.59　成功连接

串口连接成功后，连接 USB 下载线。标题栏显示 USB:OK 后，按住空格键重新启动开发板进入 VIVI 模式，如图 2.60 所示。然后可以进行分区、重新下载 Bootloader、下载内核和文件系统。具体过程将在移植相关部分时进行详细说明。

注意：在第二次使用 DNW 工具时，也可能因为使用的 USB 端口与上次不同或者其他原因无法识别 USB 设备。此时会在设备管理器对话框的通用串行总线控制器中，出现一个驱动为 unknow device 的设备，如图 2.61 所示。右击该项重新安装驱动，重新安装完驱动后重新启动开发板。正确安装后设备管理器下会正确显示设备的名字，如图 2.62 所示，且 DNW 对话框中显示 USB:OK，此时可以通过 USB 端口下载内核和文件系统映像文件了。

```
DNW v0.50A   [COM4,115200bps][USB:OK]
Serial Port  USB Port  Configuration  Help

VIVI version 0.1.4 (root@vmwarelinux) (gcc version 2.95.3 20010315 (release))
#0.1.4 Mon May 14 15:46:54 CST 2007
MMU table base address = 0x33DFC000
Succeed memory mapping.
DIVN_UPLL0
MPLLVal [M:7fh,P:2h,S:1h]
CLKDIVN:5h


+---------------------------------------------------------+
|  FriendlyARM SBC2440/QQ2440  USB Downloader ver1.2  |
+---------------------------------------------------------+
USB: IN_ENDPOINT:1 OUT_ENDPOINT:3
FORMAT: <ADDR(DATA):4>+<SIZE(n+10):4>+<DATA:n>+<CS:2>
NOTE: Power off/on or press the reset button for 1 sec
      in order to get a valid USB device address.

NAND device: Manufacture ID: 0xec, Chip ID: 0x76 (Samsung K9D1208V0M)
Could not found stored vivi parameters. Use default vivi parameters.
Press Return to start the LINUX/Wince now, any other key for vivi
type "help" for help.
FriendlyARM>
FriendlyARM> bon part 0 192k 1216k
```

图 2.60　VIVI 模式

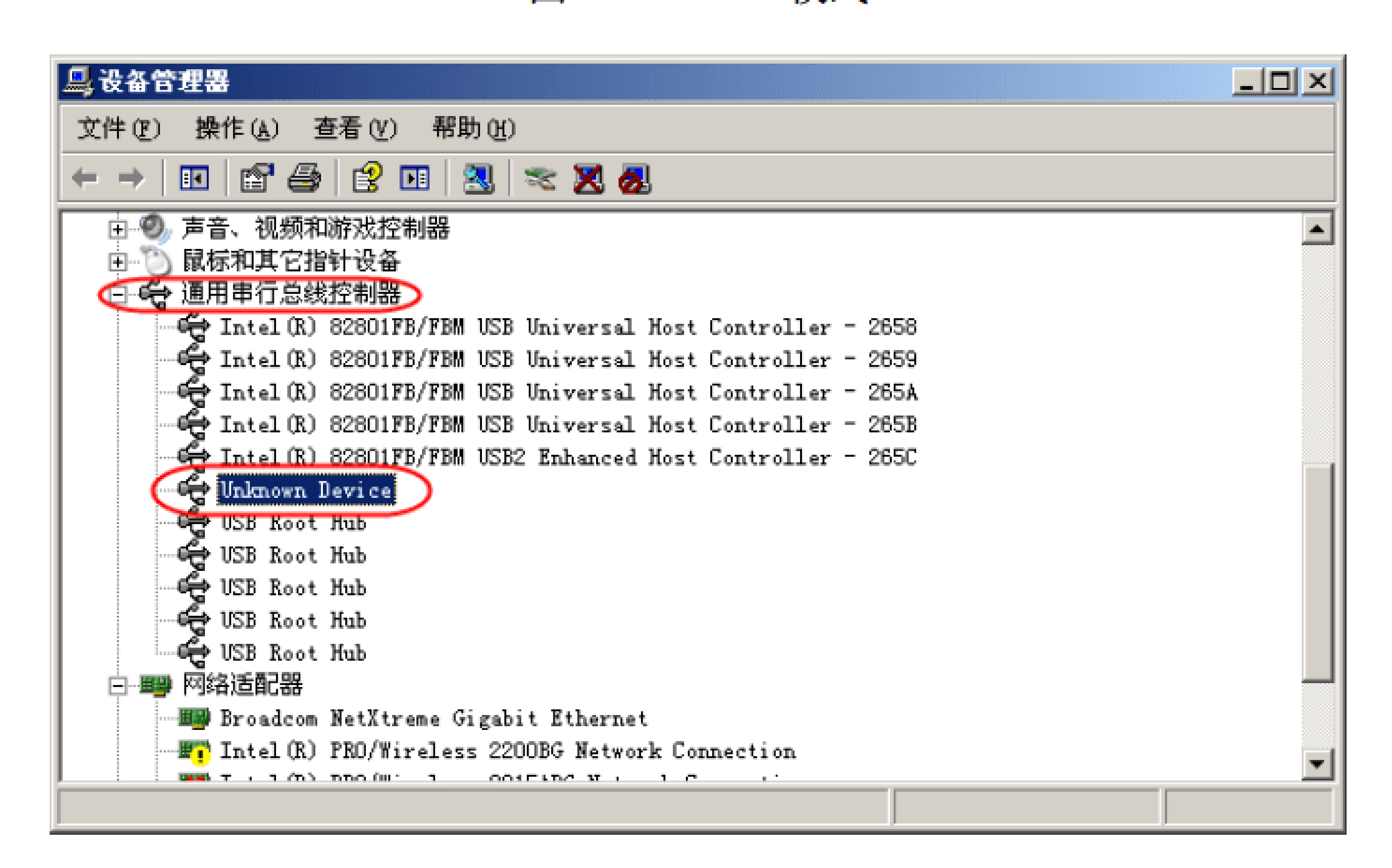



图 2.61　DNW USB 驱动无法识别

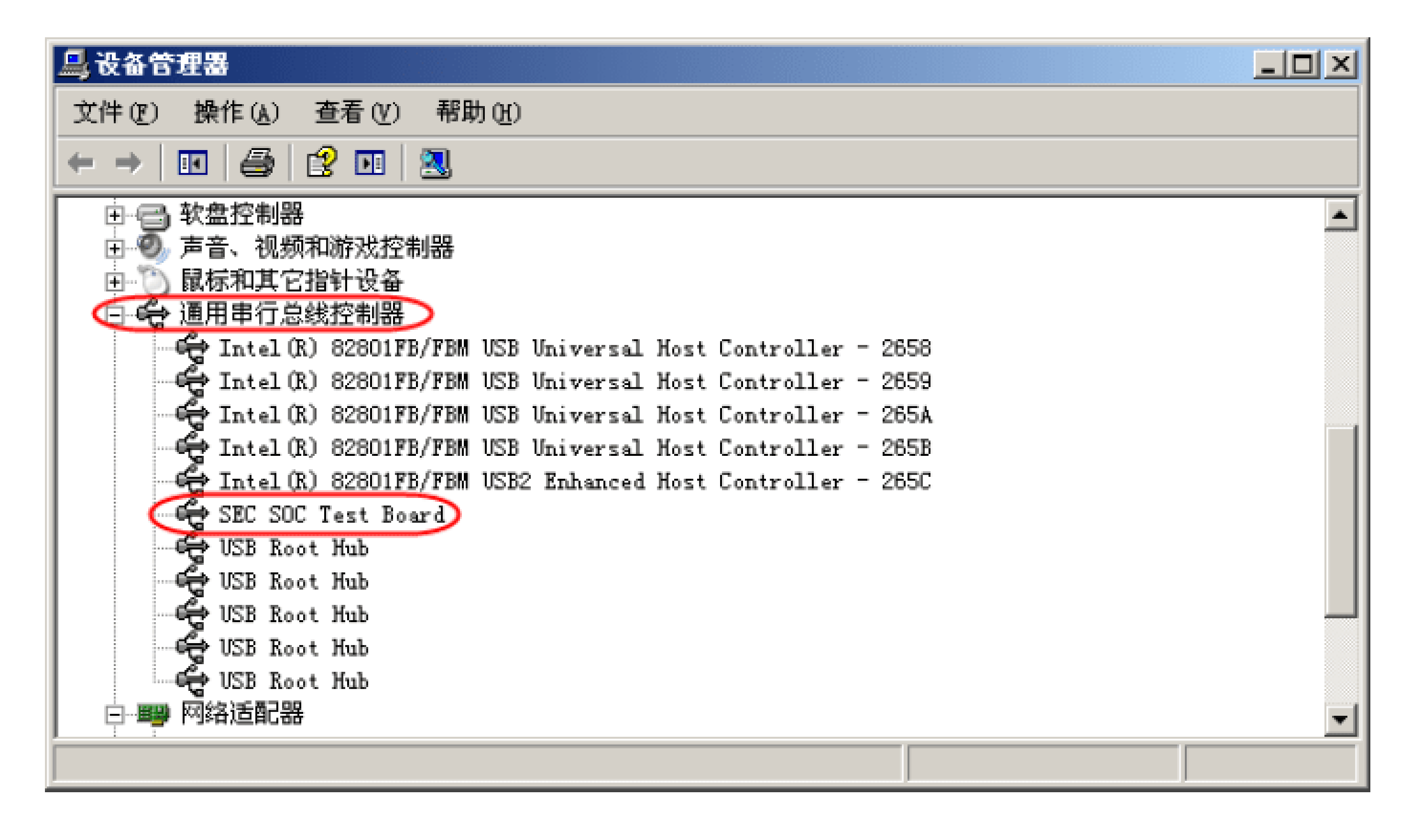



图 2.62　DNW USB 驱动重新正确安装

### 2.4.3　应用程序和文件传输

整个系统移植成功后，还有一些应用程序或者文件要在开发板和上位机之间进行传输。一般选择 FTP 进行传输。在上位机上安装 FTP 服务器软件，启动服务器。

启动超级终端，查看开发板是否设置了与上位机相同 IP 段的 IP 地址，如果没有进行设置，则使用 ifconfig 进行设置，设置后查看如图 2.63 所示。

```
#ifconfig eth0 192.168.1.230
#ifconfig
```

```
2440 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
[root@FriendlyARM /]# ifconfig eth0 192.168.1.230
[root@FriendlyARM /]# ifconfig
eth0      Link encap:Ethernet  HWaddr 08:90:90:90:90:90
          inet addr:192.168.1.230  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:68 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8446 (8.2 KiB)  TX bytes:0 (0.0 B)
          Interrupt:51

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

[root@FriendlyARM /]#
已连接 0:04:1 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

图 2.63　设置开发板 IP 地址

使用 FTP 命令登录上位机前，测试能否与上位机建立正确的连接，如图 2.64 所示。建立正确的连接后，使用建立 FTP 服务器时设置的用户名和密码进行登录。登录过程如图 2.65 所示。

```
#Ping 192.168.1.199
#ftp 192.168.1.199
Name:root                //提示输入用户名"root"，在服务器上建立的用户名
Password:123             //提示输入密码"123"，在服务器上建立 root 用户的密码
ftp>binary               //传输文件时使用的格式
```

使用 ls 可以显示服务器共享的文件，使用 get 命令可以从服务器上下载文件，使用 put 命令可以向服务器上传文件。完成文件传输可以使用 bye 退出 FTP，如图 2.66 所示。

```
ftp>ls
ftp>get usb_modeswitch.conf /etc/usb_modeswitch.conf
                                   //从服务器上下载文件 usb modeswitch.conf
ftp>bye                            //退出 ftp
#ls /etc/
```

```
2440 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
[root@FriendlyARM /]# ping 192.168.1.199
PING 192.168.1.199 (192.168.1.199): 56 data bytes
64 bytes from 192.168.1.199: seq=0 ttl=128 time=4.194 ms
64 bytes from 192.168.1.199: seq=1 ttl=128 time=0.538 ms
64 bytes from 192.168.1.199: seq=2 ttl=128 time=0.470 ms
64 bytes from 192.168.1.199: seq=3 ttl=128 time=0.518 ms
64 bytes from 192.168.1.199: seq=4 ttl=128 time=0.520 ms
64 bytes from 192.168.1.199: seq=5 ttl=128 time=0.524 ms
64 bytes from 192.168.1.199: seq=6 ttl=128 time=0.521 ms
64 bytes from 192.168.1.199: seq=7 ttl=128 time=0.522 ms
64 bytes from 192.168.1.199: seq=8 ttl=128 time=0.520 ms
64 bytes from 192.168.1.199: seq=9 ttl=128 time=0.523 ms
64 bytes from 192.168.1.199: seq=10 ttl=128 time=0.524 ms
64 bytes from 192.168.1.199: seq=11 ttl=128 time=0.466 ms
64 bytes from 192.168.1.199: seq=12 ttl=128 time=0.467 ms
64 bytes from 192.168.1.199: seq=13 ttl=128 time=0.524 ms
_
已连接 0:01:0 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

图 2.64　检验开发板是否与上位机建立连接

```
2440 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
[root@FriendlyARM /]# ftp 192.168.1.199
Connected to 192.168.1.199.
220 Serv-U FTP Server v6.0 for WinSock ready...
Name (192.168.1.199:root):root
331 User name okay, need password.
Password:
230 User logged in, proceed.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> binary
200 Type set to I.
ftp> _
已连接 0:03:0 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

输入密码

图 2.65　登录 FTP 服务器过程

```
2440 - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
ftp> get usb_modeswitch.conf /etc/usb_modeswitch.conf
local: /etc/usb_modeswitch.conf remote: usb_modeswitch.conf
200 PORT Command successful.
150 Opening BINARY mode data connection for usb_modeswitch.conf (28914 Bytes).
226 Transfer complete.
28914 bytes received in 1.53 secs (18 Kbytes/sec)
ftp> bye
221 Goodbye!
[root@FriendlyARM /]# ls /etc/
boa                  init.d               pointercal
eth0-setting         issue.net            profile
ftpchroot            localtime            protocols
ftpusers             login.defs           rc.d
group                mdev.conf            resolv.conf
host.conf            mime.types           scsi_id.config
hosts                mtab                 services
inetd.conf           passwd               usb_modeswitch.conf
[root@FriendlyARM /]#
已连接 0:24:0 ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印
```

图 2.66　传输文件过程

# 2.5　在开发中使用网络文件系统（NFS）

在开发中使用 NFS 有两个优点：开发过程中不受开发板空间的限制，直接使用网络文件就像使用本地文件一样；调试过程中避免一一将编译后的应用程序和库文件复制到开发板上。在开发板中使用网络文件系统可以为开发和调试节省不少时间。下面介绍虚拟机环境下搭建 NFS 的过程。

## 2.5.1　虚拟机设置

这里需要配置虚拟机，让虚拟机能够直接访问局域网内的任何主机。前面为了能够让虚拟机与宿主机进行通信，将虚拟机的网络连接设置为 NAT 方式，下面主要介绍桥接模式和 NAT 模式。

### 1．桥接模式（Bridged Networking）

在桥接模式下，VMWare 虚拟出来的操作系统相当于局域网中一台独立的主机，它可以访问网内任何一台机器。在这种模式下，需要手动为虚拟系统配置和宿主机器处于同一网段的 IP 地址和子网掩码，这样虚拟系统就可以和宿主机器进行通信。如果配置好网关和 DNS 的地址，还可以通过局域网的网关或路由器访问互联网。

### 2．NAT模式（Network Address Translation）

在 NAT 模式下，虚拟系统借助 NAT（网络地址转换）功能，通过宿主机器所在的网络来访问互联网。NAT 模式下的虚拟系统的 TCP/IP 配置信息是由 VMnet（NAT）虚拟网络的 DHCP 服务器提供，无法进行手动修改，因此虚拟系统和局域网中的其他真实主机无法通信。

为了使得虚拟机、宿主机和开发板能达到互相通信的目的，虚拟机的网络连接方式应该采用桥接方式，通过选择菜单“虚拟机”，然后在下拉菜单中选择“设置”选项，在弹出的 Virtual Machine Setting 窗口中进行设置，如图 2.67 所示。

**注意：** 设置虚拟机网络连接时，应该是在虚拟机没有启动时进行设置，否则无法设置或者设置无法生效。

## 2.5.2　虚拟机的 IP 地址设置

启动虚拟机，查看虚拟机的 IP 地址和网络连接状态。在右下角查看虚拟网卡是否连接，在终端输入 ifconfig 查看 eth0 是否设置，如图 2.68 所示。

或者可以通过“系统”|“管理”|“网络”命令打开“网络配置”对话框，选择“设备”选项卡，如图 2.69 所示。双击设备 eth0 进入以太网设置窗口，在该窗口中对虚拟机 IP 地址和网关进行设置，如图 2.70 所示。

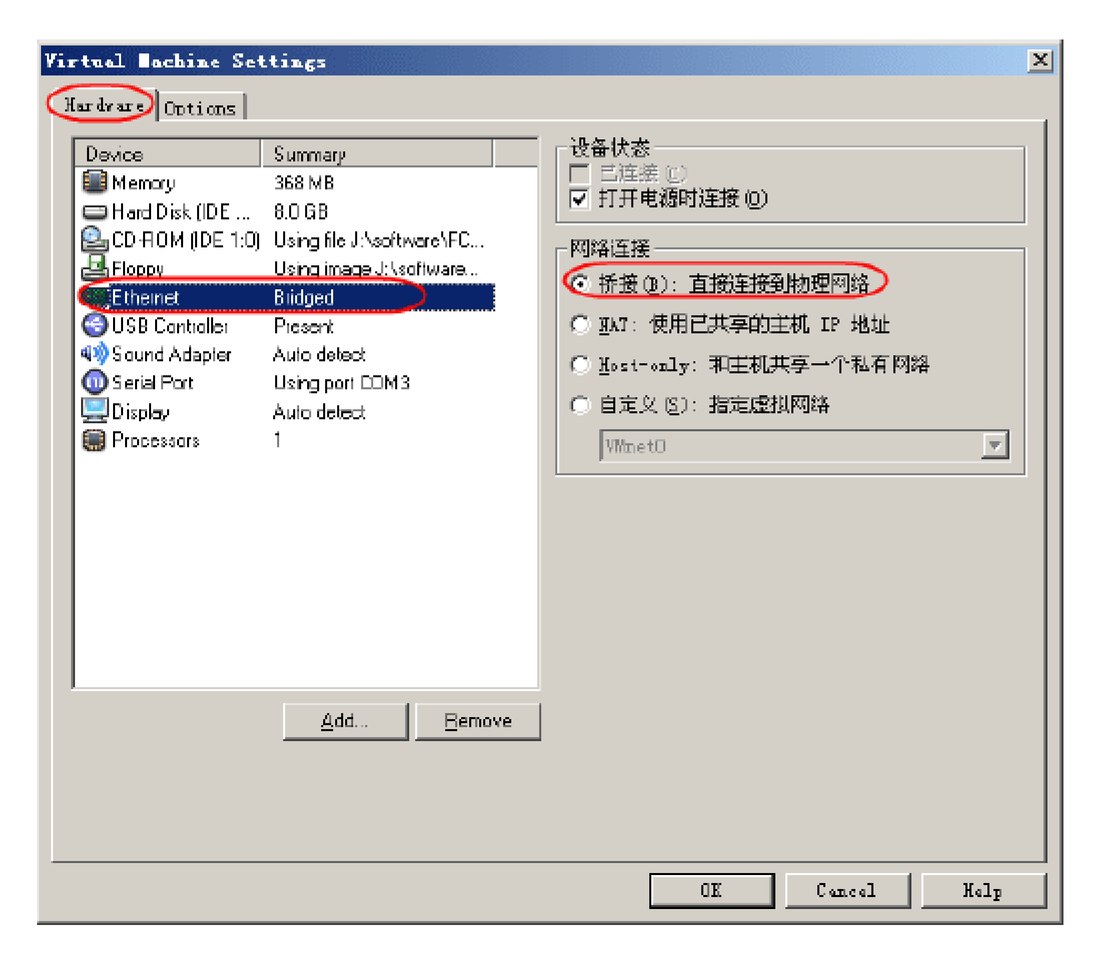



图 2.67　网络连接方式为桥接方式

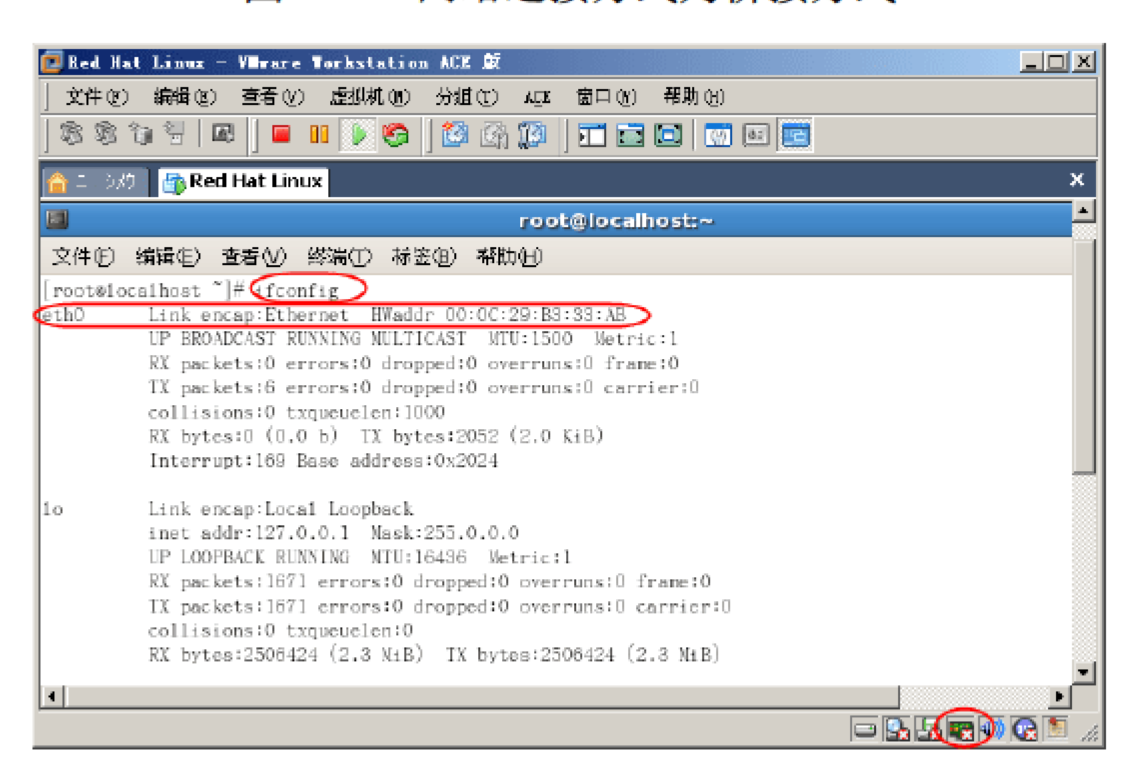
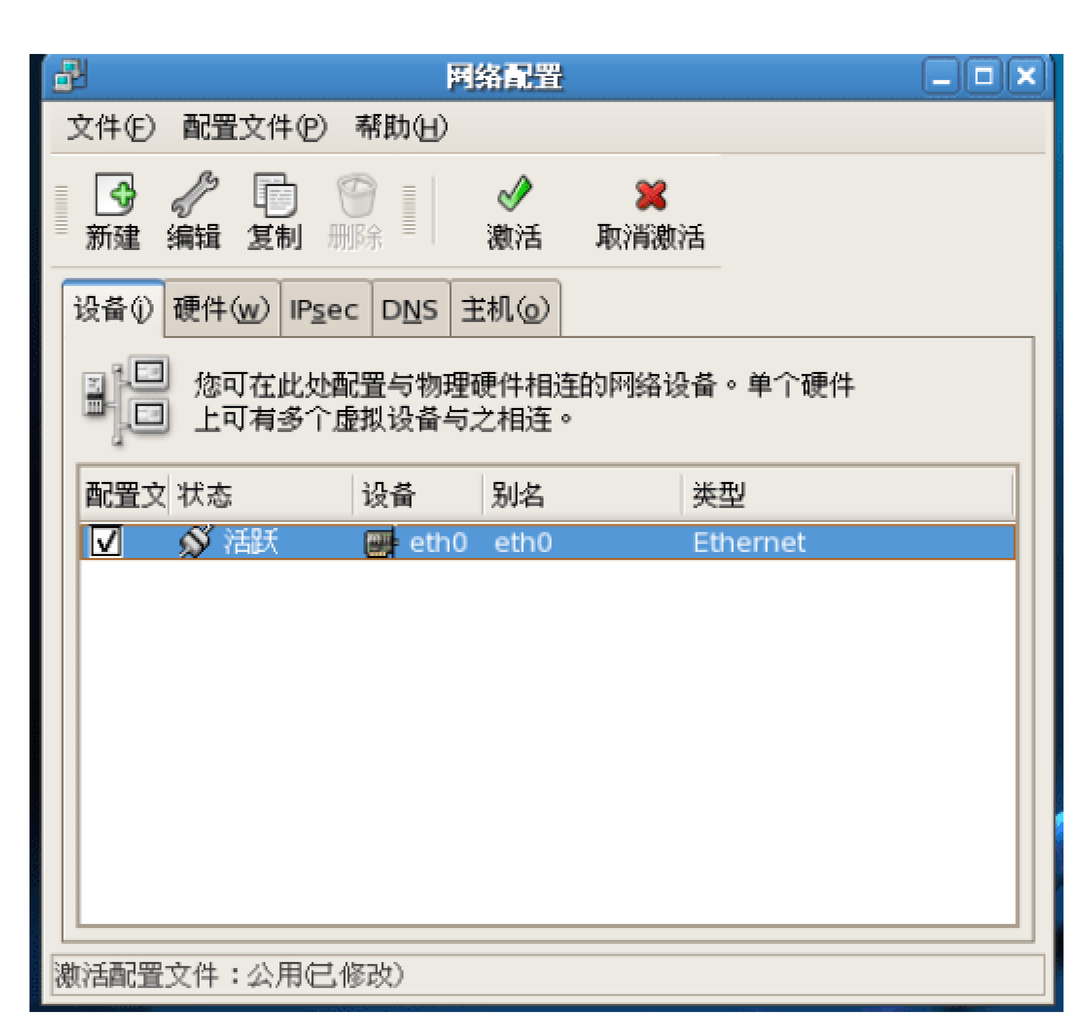



图 2.68　未设置前虚拟机的网络连接状态



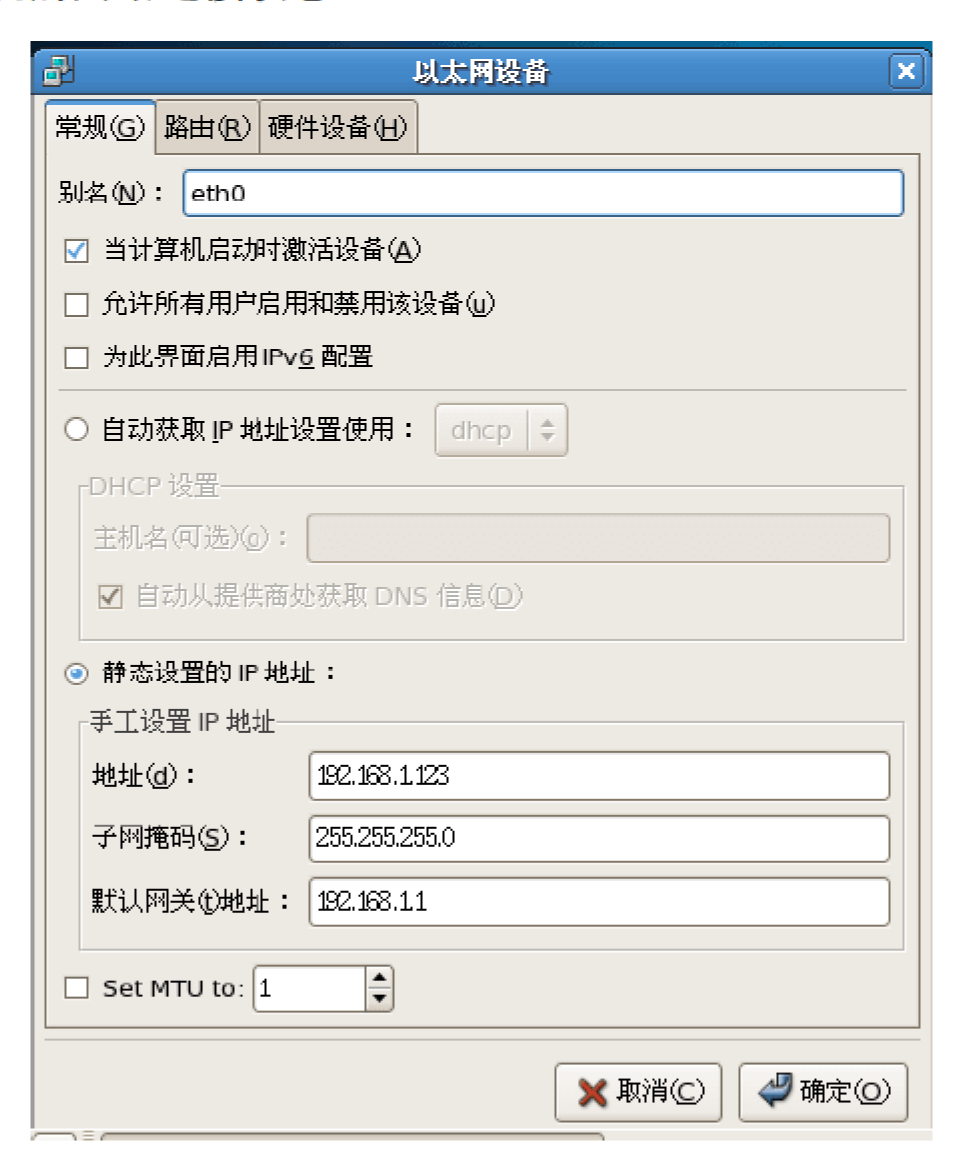




图 2.69　虚拟机网络配置　　　　图 2.70　虚拟机 IP 设置

注意：虚拟机启动后看虚拟机右下角的虚拟网卡标识是否连接上，如果没有连接上则有可能是安装虚拟机时少选了一项 VMware Bridge Protocol。如果连接上就不必进行下面的安装过程了。

可以打开主机网络连接的属性窗口，在该窗口中有本地连接 VMnet1 和 VMnet8 和网络连接。右击 VMnet1 或者 VMnet8，选择“属性”进入“VMware Network Adapter VMnet1 属性”对话框，单击“安装”按钮，如图 2.71 所示。进入“选择网络组件类型”对话框后，选择“服务”选项，并单击“添加”按钮，如图 2.72 所示。

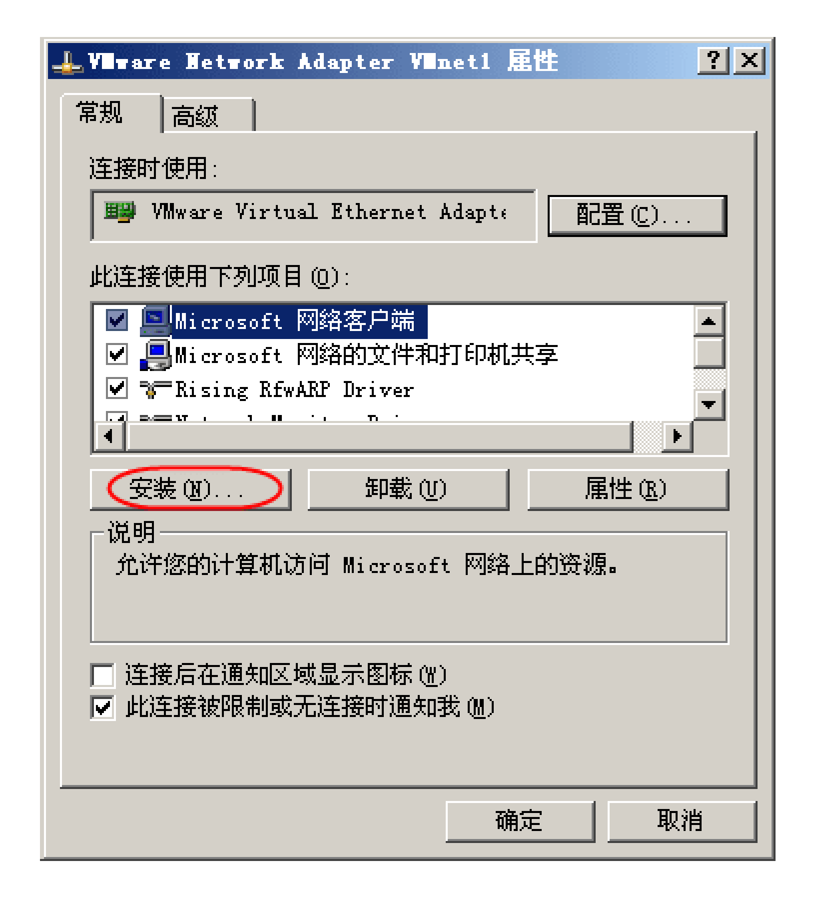



图 2.71　进入 VMware Network Adapter VMnet1 属性窗口

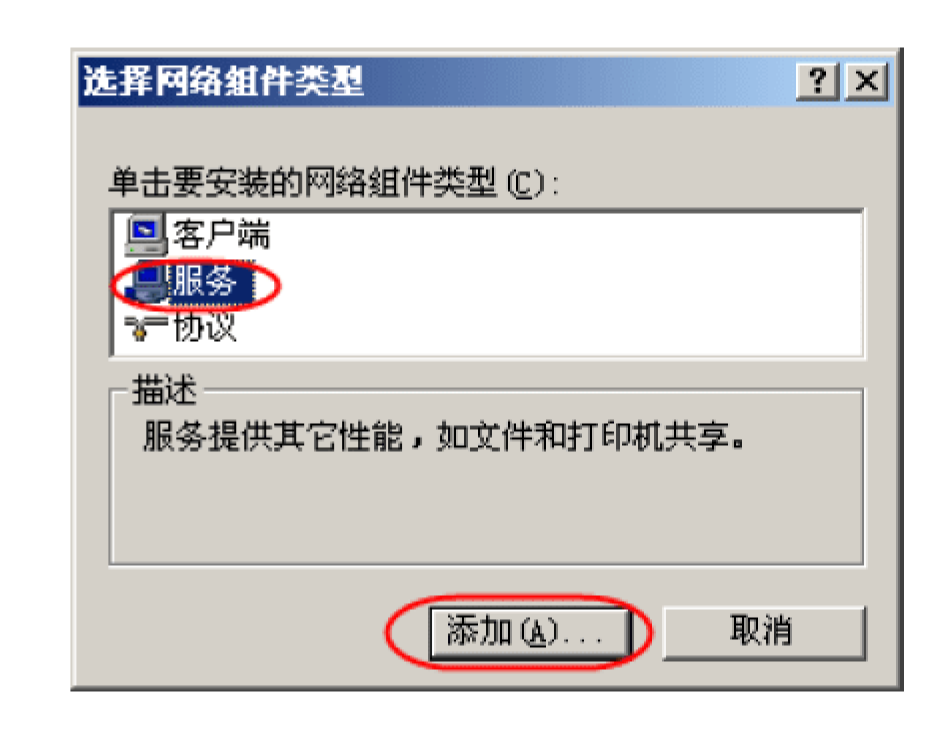



图 2.72　添加服务

在选择网络客户端窗口，单击“从磁盘安装”，如图 2.73 所示。在虚拟机安装路径下的 VMware Workstation 目录中找到 netbridge.inf 文件，选择“打开”按钮进行安装，如图 2.74 所示。

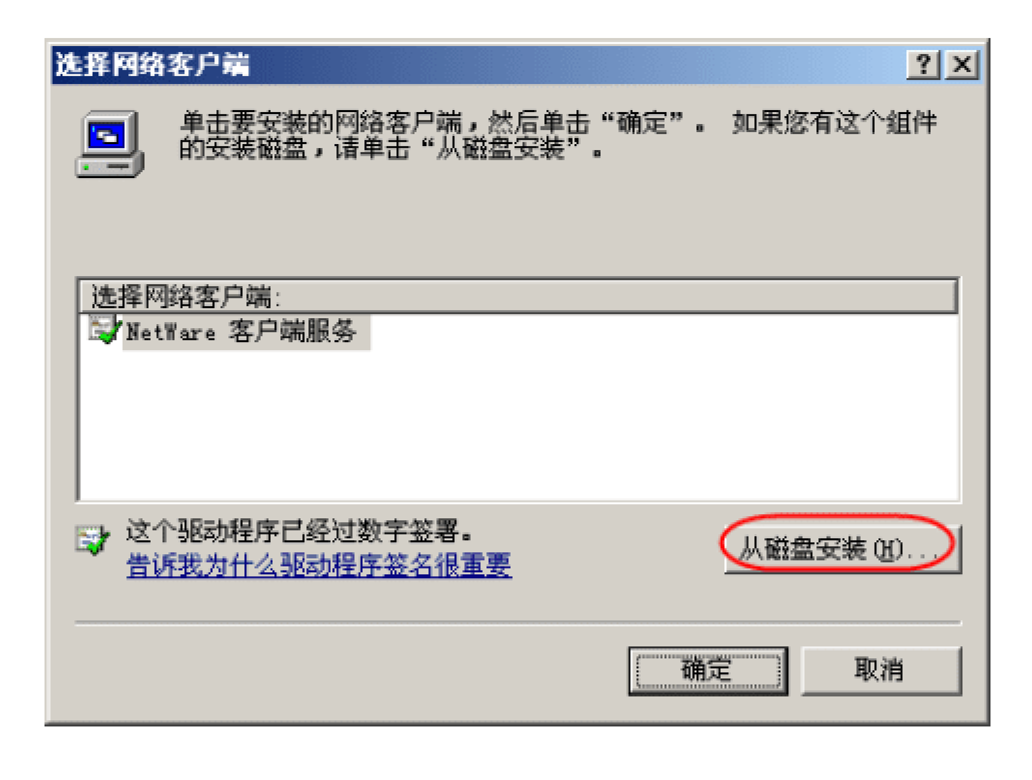



图 2.73　选择从磁盘安装

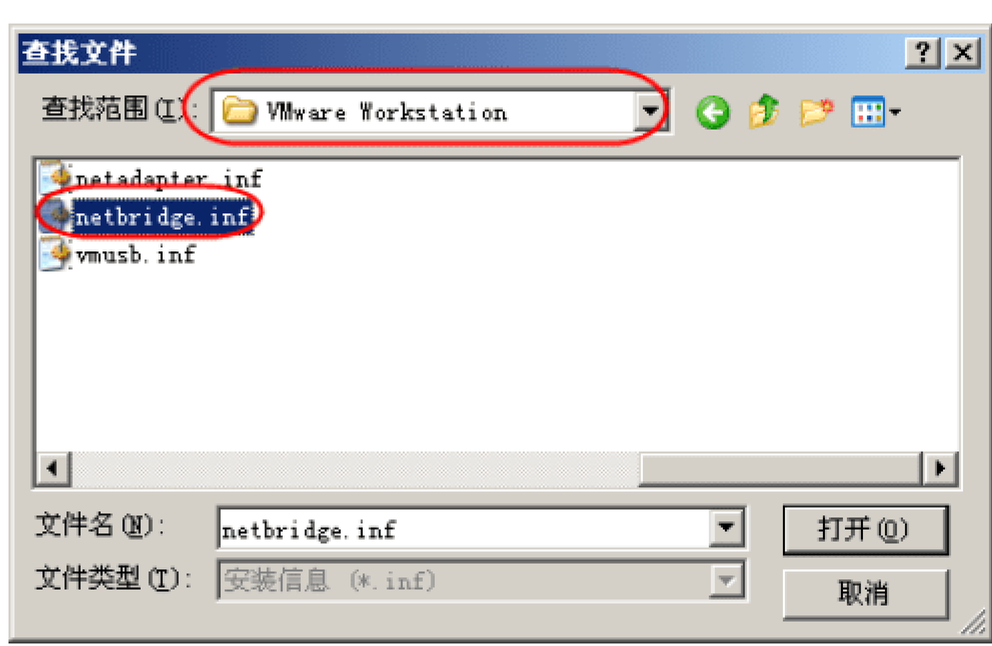



图 2.74　打开安装文件 netbridge.inf

安装过程完成后，在 VMware Network Adapter VMnet1 属性窗口出现了 VMware Bridge Protocol 项，如图 2.75 所示。重新启动计算机，并且重启虚拟机。

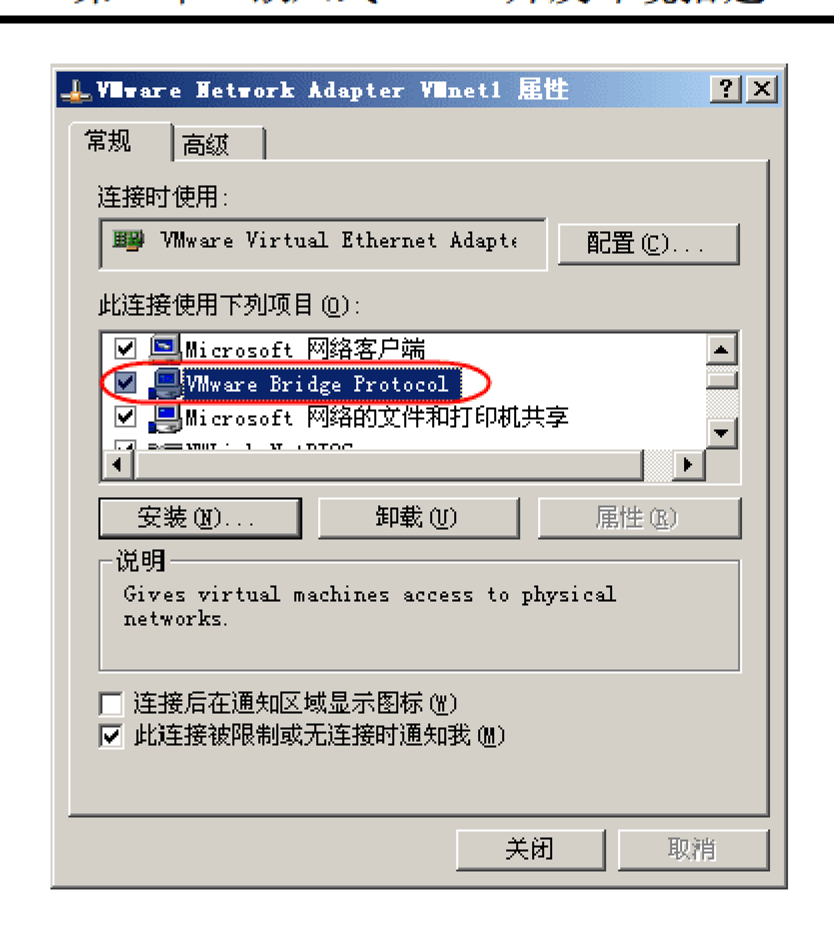



图 2.75　安装后出现 VMware Bridge Protocol 项

## 2.5.3　验证网络连接

主机的 IP 地址为 192.168.1.199，虚拟机的 IP 地址为 192.168.1.123，开发板的 IP 地址为 192.168.1.230。分别通过 ping 命令验证两两之间是否可以通信，正常情况下是可以互相 ping 通，但是如果物理网卡没有连接网线则无法实现通信（开发板与主机采用交叉网线连接），如图 2.76 是虚拟机 ping 主机的情况。

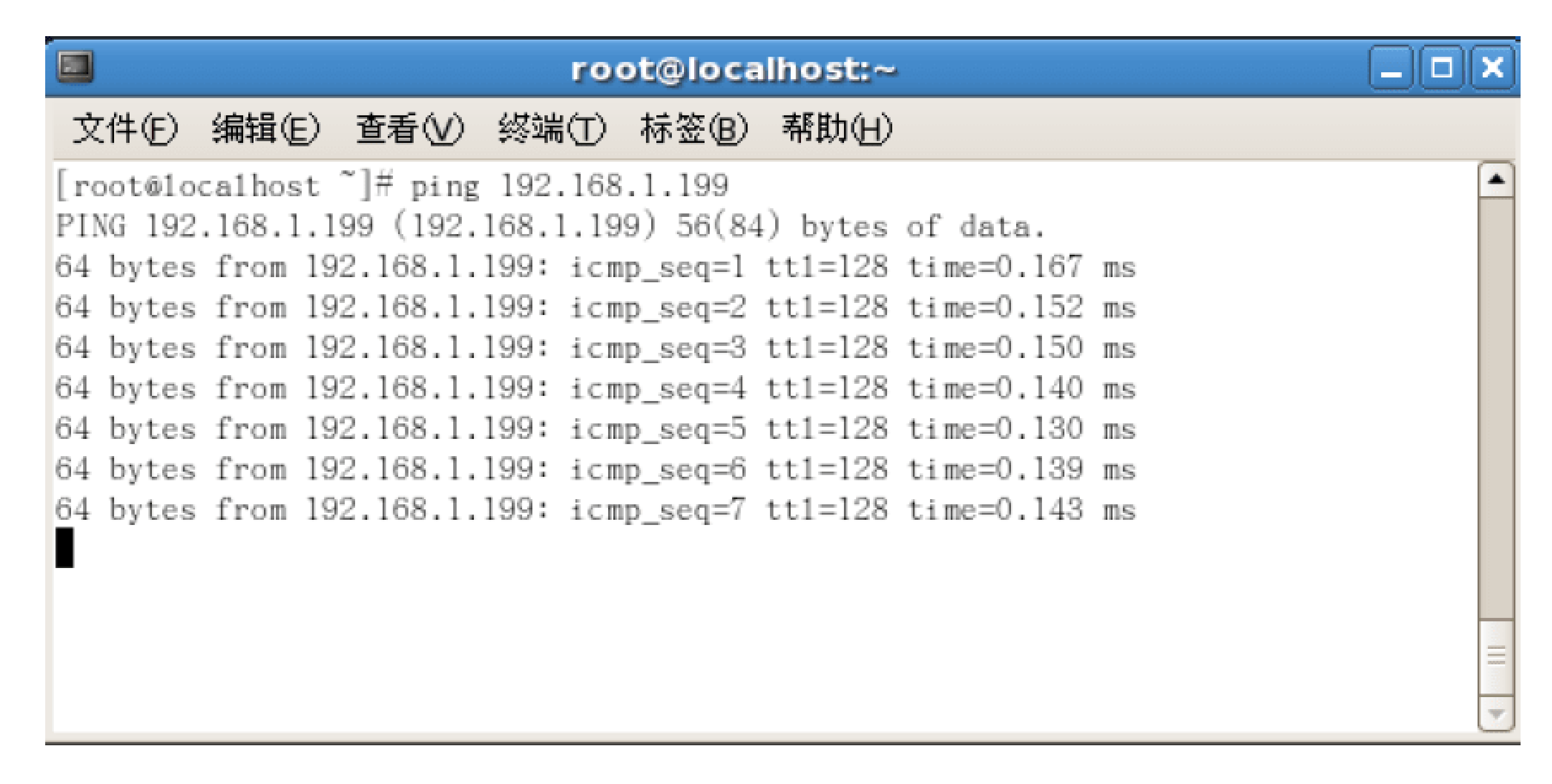



图 2.76　虚拟机 ping 主机

## 2.5.4　设置共享目录

编辑文件/etc/exports，在文件中添加以下内容。

```
/home/nfs 192.168.1.*(rw, sync, no_root_squash)
```

- /home/nfs：表示共享给其他主机的共享目录；

- 192.168.1.*：表示 IP 地址为 192.168.1.2/254 的主机都能够挂载/home/nfs 目录；
- rw：表示挂接此目录的客户机对该目录具有读写的权力；
- sync：表示同步写入存储器；
- no_root_squash：表示允许挂接此目录的客户机享有该主机的 root 身份。

使用下面的命令查看共享目录：

```
# showmount -a
```

如果出现错误：showmount: can't get address for localhost.localdomain，则修改文件/etc/hosts，将

```
::1      localhost.localdomain   localhost
```

修改为：

```
127.0.0.1      localhost.localdomain   localhost
```

## 2.5.5　启动 NFS 服务

启动 NFS 服务之前，先关闭防火墙，并且启动 PortMap 服务。在终端中输入 setup 命令，进入选择一种工具界面，如图 2.77 所示。使用 Tab 键选择“运行工具”回车确定，进入防火墙配置禁止防火墙，如图 2.78 所示。




图 2.77　选择工具界面




图 2.78　禁止防火墙

启动 PortMap 服务和 NFS 服务的命令如下：

```
# service portmap start     或  # /etc/init.d/portmap start     //服务启动
# service portmap restart   或  # /etc/init.d/portmap restart   //服务重启
# service nfs start         或  # /etc/init.d/nfs start
# service nfs restart       或  #/etc/init.d/nfs restart
```

如果在启动过程中出现：下面“启动 NFS 守护进程失败”等错误，则重新计算机后，再进行尝试。

正确启动 NFS 服务过程和信息如下：

```
# service portmap restart
停止 portmap:                                          [确定]
```

```
启动 portmap:                                           [确定]
# service nfs restart
关闭 NFS mountd:                                        [确定]
关闭 NFS 守护进程:                                      [确定]
关闭 NFS quotas:                                        [确定]
关闭 NFS 服务:                                          [确定]
启动 NFS 服务:                                          [确定]
关掉 NFS 配额:                                          [确定]
启动 NFS 守护进程:                                      [确定]
启动 NFS mountd:                                        [确定]
```

### 2.5.6　修改共享配置后

修改/etc/exports 配置文件后，应该使配置文件重新生效。在启动了 NFS 之后又修改了/etc/exports，此时就可以用 exportfs 命令使改动立刻生效，该命令格式如下：

```
exportfs [-aruv]
```

- -a：全部 mount 或者 unmount /etc/exports 中的内容；
- -r：重新 mount /etc/exports 中共享出来的目录；
- -u：umount 目录；
- -v：在 export 的时候，将详细的信息输出到屏幕上。

例如：

```
exportfs -rv
```

该命令重新输出全部的共享目录信息。在每次修改了/etc/exports 文件后都要运行一次该命令，使共享配置生效。

### 2.5.7　挂载 NFS

在虚拟机上修改共享目录/home/nfs 的权限为 777。开发板与主机通过交叉网线连接后，虚拟机、主机及开发板三者可以进行互相通信。使用 mount 命令在开发板上挂载此目录。

```
#chmod 777 /home/nfs
# mount -t nfs 192.168.1.123:/home/nfs /mnt
或者使用
# mount -o nolock -t nfs 192.168.1.123:/home/nfs /mnt
```

### 2.5.8　双网卡挂载 NFS

当拥有两张物理网卡时，专门用一张网卡将 ARM 板和虚拟机相连，将两者的 IP 设置在一个 IP 段内。具体过程和单网卡类似，首先做到虚拟机和 ARM 能相互 ping 通，接着启动 NFS 服务正常，最后挂载网络文件系统。

在搭建 NFS 时，给出一些错误情况解决的方法。

- 当启动 NFS 服务失败时，解决的办法通常是修改/etc/exports 文件，出错的原因通

常是权限引起的。

- 当出现 RPC 等报错时，应该注意防火墙是否关闭。
- 当挂载 NFS 时，出现 Permission denied 报错时，检查/etc/exports 文件中的权限设置，另外检查共享目录的权限设置。

出现任何报错的情况，都应该查看错误日志/var/log/messages，对照错误日志查找问题。笔者在挂载的过程中遇到几个问题。

（1）mount：RPC：Timed out

该问题是由主机的防火墙引起的，关闭了虚拟机的防火墙后请注意，主机的防火墙也可能对 RPC 的包进行拦截。遇到此类问题时，请读者注意虚拟机和主机两者的防火墙是否关闭。

（2）在使用下面的 mount -t nfs 192.168.1.123:/home/nfs /mnt 命令进行挂载时，出现下面的错误。

```
rpcbind: server localhost not responding, timed out
RPC: failed to contact local rpcbind server (errno 5).
rpcbind: server localhost not responding, timed out
RPC: failed to contact local rpcbind server (errno 5).
rpcbind: server localhost not responding, timed out
RPC: failed to contact local rpcbind server (errno 5).
将挂载命令修改如下后挂载成功
#mount -o nolock 192.168.1.123:/home/nfs /mnt    或者
#mount -o nolock -t nfs 192.168.1.123:/home/nfs /mnt
```

# 2.6　小　　结

安装交叉编译工具是本章最复杂的一节，建议读者在有充足的时间及有必要的情况下才去编译，一般可以直接使用开发板公司提供的稳定的交叉编译工具。另外在编译前，应该参照一些编译成功的例子，与其版本最好一致。如果编译过程中出现问题，首先应该考虑版本是否兼容。更换新版本时应该清除旧版本编译过程中生成的文件。重新编译后应该核对生成工具的版本是否和更换的版本相符。因为初次编译交叉环境需要很长时间，所以在继续编译时有可能出现环境变量失效的问题。读者在安装的过程中可以直接使用全路径或者设置永久的环境变量的方式。一般而言，对编译安装路径变量不提倡设置为永久的环境变量，因为在以后的内核编译、驱动移植、文件系统移植等编译过程中都会有路径设置情况。

# 第 2 篇　系统移植技术篇

# 第 3 章　Bootloader 移植

Bootloader 是在嵌入式系统运行之前运行的一段程序。通过这段 Bootloader 小程序，可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。本章主要介绍两种常见的 Bootloader 及其移植过程。

## 3.1　Bootloader 介绍

体系结构不同的 CPU 都有不同的 Bootloader，有些 Bootloader 支持多种不同类型体系结构的处理器，如 U-boot。通常，Bootloader 不但依赖于 CPU 的体系结构，而且依赖于特定的嵌入式板级设备的配置，即对于两块不同的嵌入式板而言，即使它们是基于同一种 CPU 而构建的，要运行在一块开发板上的 Bootloader 程序能够运行在另一块开发板上，通常需要修改 Bootloader 的源程序以适应不同的开发板。

### 3.1.1　Bootloader 与嵌入式 Linux 系统的关系

从软件的角度可将嵌入式 Linux 系统划分成 4 个层次，4 个层次由低层到高层分别如下所示。

- 引导加载程序：包括固化在固件中（firmware）中的 boot 代码（可选）和 Bootloader 两大部分。
- 内核：给具体类型开发板定制的内核及控制内核引导系统的参数。
- 文件系统：包括根文件系统和建立与 FLASH 内存设备上的文件系统。
- 用户应用程序：用户的应用程序，包括 GUI、Web 服务器、数据库、网络协议栈等。

### 3.1.2　Bootloader 基本概念

Bootloader 是在操作系统内核运行前执行的一段小程序，类似在启动 Windows 系统前运行的 BIOS 程序。通过这段小程序，完成了对必要硬件设备的初始化，创建内核需要的信息并将这些信息通过相关机制传递给内核，从而将系统的软硬件环境带到一个合适的状态，最终调用操作系统内核，起到引导和加载内核的功能。

### 1．Bootloader的安装媒介

系统每次加电或复位后，CPU 都会固定从预先设定的地址上取指令。基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备（比如 ROM、EEPROM 或 FLASH 等）被映射到这个预先设定的地址上。

一个同时安装有 Bootloader、内核的启动参数、内核映像和根文件系统映像的固态存储设备的典型空间分配结构图，如图 3.1 所示。

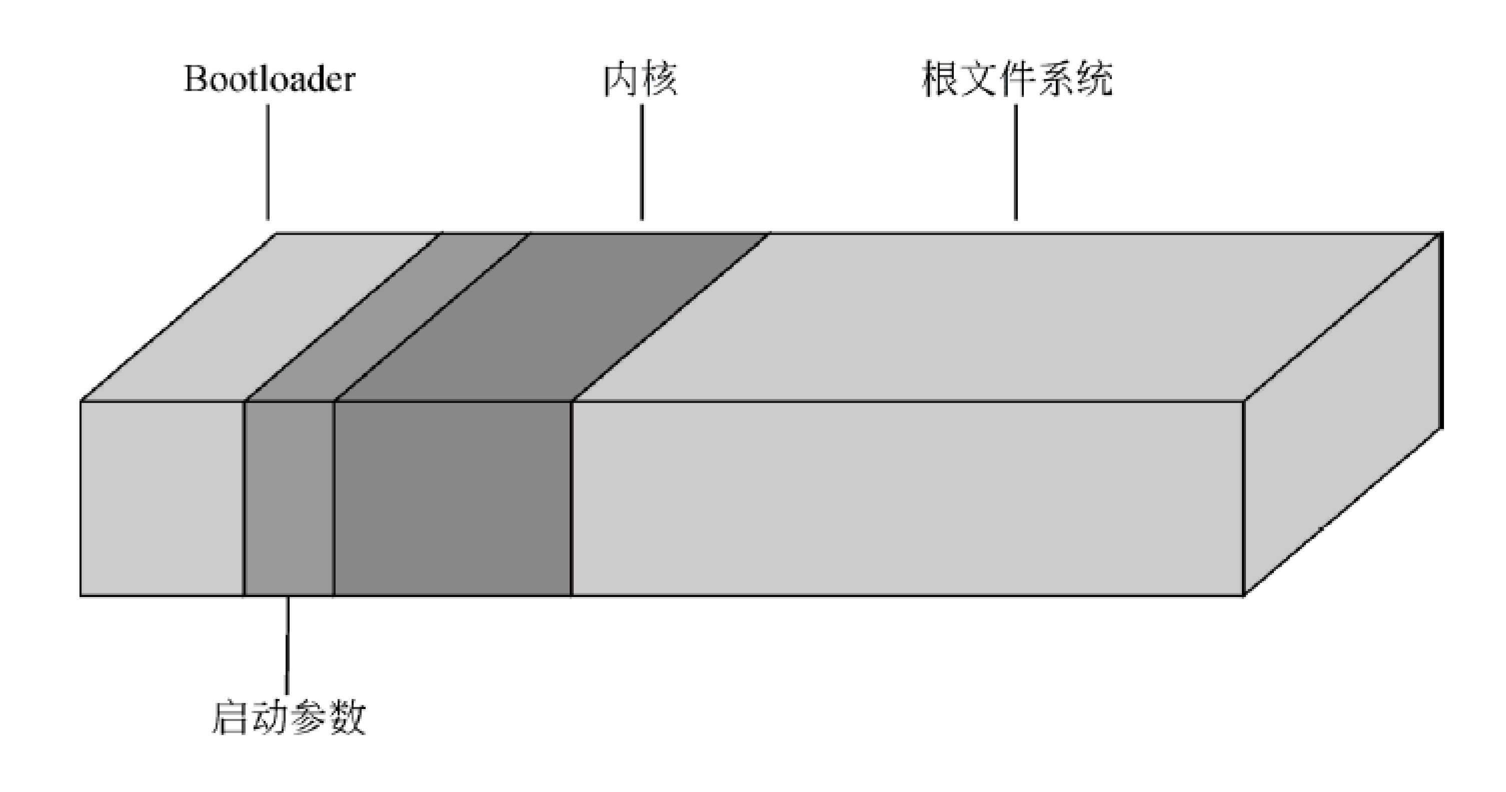



图 3.1　固态存储设备的典型空间分配结构图

### 2．Bootloader启动过程分类

Bootloader 启动过程分为单阶段和多阶段两种。相对单阶段 Bootloader 而言，多阶段的 Bootloader 则功能更加复杂，可移植性更加优越。从固态存储设备上启动 Bootloader 一般可分为两个阶段的启动过程，即启动过程可以分为 stage 1 和 stage2 两部分。

### 3．Bootloader的操作模式

绝大部分 Bootloader 均包含两种不同类型的操作模式，即“启动加载”模式和“下载”模式。

- 启动加载模式：这种模式也称为“自主”模式。即 Bootloader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。启动加载模式为 Bootloader 的正常工作模式，因此在嵌入式产品发布时，Bootloader 只能工作在该模式下。
- 下载模式：这种模式下，目标机上的 Bootloader 将通过串口连接、网络连接或 USB 连接等通信手段从主机（Host）下载文件，如下载内核映像文件和文件系统映像文件等。从主机下载的文件通常首先被 Bootloader 保存到目标板的 ROM 中，然后再被 Bootloader 写到目标板上的 FLASH 类固态存储设备中。Bootloader 的这种模式通常在第一次安装内核与根文件系统时被使用，或者在系统更新时使用。Bootloader 工作在下载模式时，通常都会提供一个命令行接口给它的终端用户，以供用户通过命令行控制 Bootloader 的工作。

### 3.1.3　Bootloader 启动过程

Bootloader 的启动过程分为 stage1 和 stage2 两个阶段，通常 stage1 是用汇编语言完成，而 stage2 则用 C 语言来实现，以便于在 stage2 阶段实现更加复杂的功能和取得更好的代码可读性及可移植性。下面介绍两个阶段分别完成的不同工作。

#### 1. stage1完成的工作

（1）基本的硬件初始化包括以下工作：

- ❑ 屏蔽所有中断。为中断提供服务通常是操作系统设备驱动程序的责任，因此在 Bootloader 的启动全过程中可以不必响应任何中断。屏蔽中断可以通过写 CPU 的中断屏蔽寄存器或状态寄存器（比如 ARM 的 CPSR 寄存器）来完成。
- ❑ 设置 CPU 速度和时钟频率。
- ❑ 初始化 RAM。包括正确设置系统内存控制器的功能寄存器，以及各内存库控制寄存器等。
- ❑ 初始化 LED。典型地，通过 GPIO 来驱动 LED，其目的是检查当前系统的状态是 OK 还是 ERROR。如果板子上没有 LED，那么也可以通过初始化 UART 向串口打印 Bootloader 的 Logo 字符信息来完成这一点。
- ❑ 关闭 CPU 内部指令和数据 cache 灯。

（2）准备 RAM 空间加载 stage2。为了获得更快的执行速度，通常把 stage2 加载到 RAM 空间中来执行，所以必须准备好一段可用的 RAM 空间范围用来加载 Bootloader 的 stage2。

（3）复制 stage2 到 RAM 中。执行复制时要确定两类地址：第一，stage2 的可执行映象在固态存储设备的存放起始地址和终止地址；第二，RAM 空间的起始地址。

（4）设置堆栈指针 sp。设置堆栈指针是为执行 C 语言代码作准备。在设置堆栈指针 sp 之前，也可以关闭 LED 灯，以提示用户我们准备跳转到 stage2。

（5）跳转到 stage2 的 C 入口点。在 ARM 处理器中，实现跳转的方法是通过修改 PC 寄存器为合适的地址。

#### 2. stage2完成的工作

（1）使用汇编语言跳转到 main()入口函数。用汇编语言写一段 trampoline 小程序，并将这段 trampoline 小程序作为 stage2 可执行映象的执行入口点。然后在 trampoline 汇编小程序中用 CPU 跳转指令跳入 main()函数中去执行；而当 main()函数返回时，CPU 执行路径再次回到 trampoline 程序。这种方法的思想是：用这段 trampoline 小程序作为 main()函数的外部包裹（external wrapper）。

（2）初始化本阶段要使用到的硬件设备。初始化至少一个串口，以便和终端用户进行 I/O 输出信息；初始化计时器等。

（3）检测系统的内存映射。所谓内存映射就是指在整个 4GB 物理地址空间中，有哪些地址范围被分配用来寻址系统的 RAM 单元。

（4）加载内核映像文件和根文件系统映像文件。包括规划内存分配布局和从 Flash 上

复制数据。规划内存分配布局包括，内核映像所占用的内存范围和根文件系统所占用的内存范围。

（5）设置内核的启动参数。在将内核映像和根文件系统映像复制到 RAM 空间中后，就可以准备启动 Linux 内核了。在调用内核之前，应该作一步准备工作，即设置 Linux 内核的启动参数。

## 3.2　Bootloader 之 U-Boot

U-Boot（全称 Universal Boot Loader）是遵循 GPL 条款的开放源码项目。从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似，实际上，在 U-Boot 的源码中很多是相应的 Linux 内核源程序的简化，尤其体现在一些设备驱动程序的源代码上，而且读者可以从 U-Boot 源码的注释中能够直接看到源码来自 Linux 内核源码。

U-Boot 不仅能够引导加载嵌入式 Linux 系统，还能引导加载 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 嵌入式操作系统。其目前要支持的目标操作系统是 OpenBSD、NetBSD、FreeBSD、4.4BSD、Linux、SVR4、Esix、Solaris、Irix、SCO、Dell、NCR、VxWorks、LynxOS、pSOS、QNX、RTEMS 和 ARTOS。U-Boot 中 Universal 的一方面是指支持前面提到的通用操作系统，另一方面是指 U-Boot 支持通用的处理器，包括 PowerPC、MIPS、x86、ARM、NIOS 和 XScale 等不同体系结构的常用系列处理器。

对大多数操作系统的支持和大多数处理的支持是 U-Boot 项目的开发目标。从目前情况来看，U-Boot 对 PowerPC 系列处理器的支持最丰富，对 Linux 操作系统的支持最完善。

### 3.2.1　U-Boot 优点

U-Boot 在目前的嵌入式开发中被广泛采用，是因为其具有很多优点。其优点包括以下几点：

- 开放源码；
- 支持多种嵌入式操作系统内核，如 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS；
- 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale；
- 较高的可靠性和稳定性；
- 较高的可靠性和稳定性；
- 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求、产品发布等；
- 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；
- 较为丰富的开发调试文档与强大的网络技术支持。

### 3.2.2　U-Boot 的主要功能

U-Boot 的功能非常强大，其主要功能如下。

- 系统引导：支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统，支持 NFS 挂载，从 FLASH 中引导压缩或非压缩系统内核。
- 基本辅助功能：强大的操作系统接口功能；可灵活设置、传递多个关键参数给操作系统，满足系统不同阶段的开发调试和产品发布要求，特别是对 Linux 支持最完善；支持目标板环境参数多种存储方式，如 FLASH、NVRAM 和 EEPROM；CRC32 校验，可校验 FLASH 中内核、RAMDISK 镜像文件是否完好。
- 设备驱动：串口、SDRAM、FLASH、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI 和 RTC 等驱动支持。
- 上电自检功能：SDRAM、FLASH 大小自动检测，SDRAM 故障检测，CPU 型号。
- 特殊功能：XIP 内核引导。

### 3.2.3　U-Boot 目录结构

以 U-Boot-1.1.6 为例介绍其目录结构。共有 27 个文件可以分为 3 类。

- 第 1 类目录与处理器体系结构或者开发板硬件直接相关；
- 第 2 类目录是一些通用的函数或者驱动程序；
- 第 3 类目录是 U-Boot 的应用程序、工具或者文档。

以 U-Boot-1.1.6 为例，其顶层目录下各级子目录的存放规则，如表 3.1 所示。

表 3.1　U-Boot各级子目录存放规则

| 目　　录 | 特　　性 | 说　　明 |
|---|---|---|
| board | 平台依赖 | 存放已有开发板相关的目录文件，例如 RPXlite（mpc8xx）、smdk2410（arm920t）和 sc520_cdp（x86）等目录 |
| cpu | 平台依赖 | 存放与 CPU 相关的目录文件，例如 mpc8xx、ppc4xx、arm720t、arm920t、xscale 和 i386 等目录 |
| lib_arm | 平台依赖 | 存放 ARM 体系结构通用的相关文件，主要是实现 ARM 平台通用的函数 |
| lib_avr32 | 平台依赖 | avr32 系统结构通用的文件 |
| lib_blackfin | 平台依赖 | 存放对 ADI Blackfin6 体系结构通用的文件，主要用于实现 ADI Blackfin 平台通用的函数 |
| lib_generic | 平台依赖 | 通用库函数的实现 |
| lib_i386 | 平台依赖 | 存放 X86 体系结构下通用的相关文件，主要是实现 X86 平台通用的函数 |
| lib_m68k | 平台依赖 | 存放对 m68k 体系结构通用的文件，主要用于实现 m68k 平台通用的函数 |
| lib_microblaze | 平台依赖 | 存放对赛灵思公司 32 位 MicroBlaze 处理架构体系结构通用的文件，主要用于实现 MicroBlaze 平台通用的函数 |
| lib_mips | 平台依赖 | 存放对 MIPS 体系结构通用的文件，主要用于实现 MIPS 平台通用的函数 |
| lib_nios | 平台依赖 | 存放对 NIOS 体系结构通用的文件，主要用于实现 NIOS 平台通用的函数 |
| lib_nios2 | 平台依赖 | 存放对 Nios-II 体系结构通用的文件，主要用于实现 Nios-II 平台通用的函数 |

续表

| 目　录 | 特　性 | 说　明 |
|---|---|---|
| lib_ppc | 平台依赖 | 存放 PowerPC 体系结构通用的相关文件，主要是实现 PowerPC 平台通用的函数 |
| nand_spl | 通用 | Nand Flash boot 的程序 |
| net | 通用 | 存放网络的程序，BOOTP 协议、TFTP 协议、RARP 协议和 NFS 文件系统的实现 |
| post | 通用 | 存放上电自检程序 |
| rtc | 通用 | RTC 的驱动程序 |
| tools | 工具 | 存放制作 S-Record 或者 U-Boot 格式的映像等工具，例如 mkimage、crc 等 |
| common | 通用 | 通用的多功能函数实现 |
| disk | 通用 | 硬盘接口程序 |
| doc | 文档 | 存放开发帮助相关文档 |
| drivers | 通用 | 存放通用的设备驱动程序，主要有以太网接口的驱动 |
| dtt | 通用 | 存放数字温度测量器或者传感器的驱动 |
| examples | 应用例程 | 一些独立运行的应用程序的例子，例如 hello_world |
| fs | 通用 | 存放文件系统的程序 |
| include | 通用 | 存放头文件和开发板配置文件，所有开发板的配置文件都在 configs 目录下 |

## 3.3　U-Boot 移植过程

本节主要以 u-boot-2009.08 为例介绍 U-Boot 移植的主要步骤，针对的开发板是 mini2440。其他版本的 U-Boot 移植主要步骤与其类似。

### 3.3.1　环境配置

在这个阶段是编译环境的基本配置，主要完成编译环境搭建、编译目录建立、Makefile 修改等。

#### 1．建立U-Boot工作目录和mini2440开发板的配置文件

u-boot 的下载地址 http://www.filesearching.com/cgi-bin/s?t=n&q=ftp.denx.de/pub/u-boot。下载 u-boot-2009.08.tar.bz2，解压到工作目录。修改 smdk2410.h 为 mini2440.h，修改 smdk2410.c 为 mini2440.c。

```
# cp -r board/samsung/smdk2410 board/samsung/mini2440
# mv board/mini2440/smdk2410.c board/mini2440/mini2440.c
# cp include/configs/smdk2410.h include/configs/mini2440.h
```

#### 2．修改 U-Boot目录下Makefile

修改 Makefile，包括设置编译环境和建立编译配置项。如果默认的交叉编译器为 arm-linux-gcc，则不需要对编译环境进行重新设置，否则需要重新添加。在 Makefile 中将

arm-linux-替换成本机的交叉编译器。

```
# vi Makefile
?smdk2410_config
```

进入 VI 后，查找字符命令找到 smdk2410_config，修改

```
smdk2410_config :       unconfig
       @$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 samsung s3c24x0
```

为

```
mini2440 config :       unconfig
       @$(MKCONFIG) $(@:_config=) arm arm920t mini2440 samsung s3c24x0
```

以上各项参数说明如下。

- unconfig：对 unconfig 的依赖，执行 unconfig 实际上是执行清理工作。

```
unconfig:
       @rm -f $(obj)include/config.h $(obj)include/config.mk \
              $(obj)board/*/config.tmp $(obj)board/*/*/config.tmp \
              $(obj)include/autoconf.mk $(obj)include/autoconf.mk.dep
```

- arm：CPU 的架构（ARCH）
- arm920t：CPU 的类型（CPU），其对应于 cpu/arm920t 子目录。
- mini2440：开发板的型号（BOARD），对应于 board/samsung/mini2440 目录。
- samsung：开发者/或经销商（vender）。
- s3c24x0：片上系统（SOC）。

由于编译器将添加的用于 nandboot 的子函数 nand_read_ll()放到了 4K 之后造成的。对 uboot 根目录下的 Makefile 文件做如下修改。

```
__LIBS := $(subst $(obj),,$(LIBS)) $(subst $(obj),,$(LIBBOARD))
```

改为：

```
__LIBS := $(subst $(obj),,$(LIBBOARD)) $(subst $(obj),,$(LIBS))
```

### 3．修改board/samsung/mini2440/Makefile

```
vi board/samsung/mini2440/Makefile
COBJS   := smdk2410.o flash.o
```

修改为：

```
COBJS   := mini2440.o flash.o
```

使用下面的命令测试前面修改的内容是否有问题，编译成功会生成 u-boot.bin 文件。

```
#make clean
#make mini2440 config
#make
```

## 3.3.2　修改 cpu/arm920t/start.S

文件 start.S 为开发板复位后最先执行的部分，包括关闭中断、关闭快速中断和寄存器

初始化、处理器时钟设置。下面是对该文件的修改。

### 1. 注释LED跳转

下面这段代码主要设置 CPU 的模式，设置模式后，跳转到 LED 初始化部分，uboot 的这部分 LED 初始化代码是为 AT9200 写的，这里注释掉跳转语句。

```
start_code:
    /*
     * set the cpu to SVC32 mode
     */
    mrs r0,cpsr          /* mrs :将程序状态寄存器的数据传送到通用寄存器，这里是将
                         当前程序状态寄存器 cpsr 的数据读到通用寄存器 r0 中*/
    bic r0,r0,#0x1f      /*bic：位清除指令，这里是将低 5 位清除，其他位保持不变。程
                         序状态寄存器的低 8 位为：I、F、T、M4、M3、M2、M1、M0。I—
                         指定外部请求，F—指定快速中断请求，T—指定 Thumb 指令集，M4
                         到 M0 指定处理器的模式*/
    orr r0,r0,#0xd3      /*orr：逻辑或指令，r0=r0||0xd3（11010011），这里就是将
                         第 0、1、4、6、7 指定为 1，即禁止外部中断请求和快速中断请求，
                         同时设置处理器模式为（10011），即超级模式（有些文章定义为
                         管理模式）*/
    msr cpsr,r0          /*msr：将通用寄存器数据写入程序状态寄存器，这里是将设置好
                         数据的 r0 中的数据写入当前程序状态寄存器 cpsr 中*/
@   bl coloured_LED_init
@   bl red_LED_on
```

### 2. CPU频率定义

s3c2440 比 s3c2410 的频率要高，s3c2440 的频率为 405MHz，下面是增加对 s3c2440 的 CPU 频率宏定义的修改。

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
    /* turn off the watchdog */
# if defined(CONFIG_S3C2400)
#  define pWTCON        0x15300000
#  define INTMSK        0x14400008  /* Interupt-Controller base addresses
*/
#  define CLKDIVN   0x14800014  /* clock divisor register */
#else
#  define pWTCON        0x53000000
#  define INTMSK        0x4A000008  /* Interupt-Controller base addresses
*/
#  define INTSUBMSK 0x4A00001C
#  define CLKDIVN   0x4C000014  /* clock divisor register */
# endif
```

修改为：

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)|| defined(CONFIG_
S3C2440)
    /* turn off the watchdog */

# if defined(CONFIG_S3C2400)
#  define pWTCON    0x15300000
#  define INTMSK    0x14400008  /* Interupt-Controller base addresses */
```

```
#  define CLKDIVN   0x14800014  /* clock divisor register */
#else
#  define pWTCON    0x53000000
#  define INTMSK    0x4A000008  /* Interupt-Controller base addresses */
#  define INTSUBMSK 0x4A00001C
#  define CLKDIVN   0x4C000014  /* clock divisor register */
# endif
#define CLK_CTL_BASE 0x4C000000
#define MDIV_405 0x7f << 12
#define PSDIV_405 0x21
#define MDIV_200 0xa1 << 12
#define PSDIV_200 0x31
```

### 3. 中断部分修改

默认情况下是通过设置 r0 寄存器为 INTMSK 屏蔽所有中断，当 CPU 类型为 2410 或 2440 时通过 INTSUBMSK 设置，修改中断禁止部分。

```
# if defined(CONFIG_S3C2410)
    ldr r1, =0x3ff
    ldr r0, =INTSUBMSK
    str r1, [r0]
# endif
```

修改为：

```
#if defined(CONFIG_S3C2410)
    ldr r1, =0x7ff              /*根据 2410 芯片手册，INTSUBMSK 有 11 位可用*/
    ldr r0, =INTSUBMSK
    str r1, [r0]
#endif
#if defined(CONFIG_S3C2440)
    ldr r1, =0x7fff             /*根据 2440 芯片手册，INTSUBMSK 有 15 位可用，设
                                置为 0 表示这些中断服务可用，而设置为 1 表示这些中
                                断服务被屏蔽*/
    ldr r0, =INTSUBMSK
    str r1, [r0]                /*将寄存器 r1 中的数据存储到 INTSUBMSK 地址*/
#endif
```

### 4. 时钟部分修改

对于时钟设置部分，增加了 2440 的时钟设置，修改时钟设置。

```
/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */
ldr r0, =CLKDIVN
mov r1, #3
str r1, [r0]
```

修改为：

```
#if defined(CONFIG_S3C2440)
    /* FCLK:HCLK:PCLK = 1:4:8 */
    ldr r0, =CLKDIVN            /*将 CLKDIVN 保存到寄存器 r0 中*/
    mov r1, #5                  /*r1=5*/
    str r1, [r0]                /* CLKDIVN =5*/
    mrc p15, 0, r1, c1, c0, 0   /*读控制寄存器*/
```

```
    orr r1, r1, #0xc0000000     /*同步，r1=r1|0xc0000000，即将 r1 的最高两位设
                                置为 1，其余各位不变*/
    mcr p15, 0, r1, c1, c0, 0   /*写控制寄存器*/
    mov r1, #CLK_CTL_BASE
    mov r2, #MDIV_405           /*r2= MDIV_405*/
    add r2, r2, #PSDIV_405      /*r2=r2+ PSDIV_405*/
    str r2, [r1, #0x04]
#else
    /* FCLK:HCLK:PCLK = 1:2:4 */
    /* default FCLK is 120 MHz ! */
    ldr r0, =CLKDIVN
    mov r1, #3
    str r1, [r0]
    mrc p15, 0, r1, c1, c0, 0
    orr r1, r1, #0xc0000000
    mcr p15, 0, r1, c1, c0, 0
    mov r1, #CLK_CTL_BASE
    mov r2, #MDIV_200
    add r2, r2, #PSDIV_200
    str r2, [r1, #0x04]
#endif
```

### 3.3.3　添加 Nand Flash 支持

对于 Mini2440 是从 NandFlash 启动，应该将其 Flash 启动部分修改为 NandFlash 启动，另外添加对 NandFlash 的访问函数。

#### 1．将Flash启动部分改成Nand Flash启动

从 nand 复制时，不能像操作 nor 那样通过总线访问硬件，必须通过相应的控制器访问，所以复制 uboot 代码到 sdram 的操作需要用一个函数 nand_read_II()来实现，这个函数放在 board/mini2440/nand_read.c 中。

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl  cpu_init_crit
#endif
```

后面添加 Nand Flash 启动代码部分，该代码比较长，可以查看资料 U-Boot 目录下文件 cpu\arm920t\start.S 中第 221～405 行。

```
#ifdef CONFIG_S3C2440
/* Offset */
#define oNFCONF 0x00
#define oNFCONT 0x04
#define oNFCMD 0x08
#define oNFSTAT 0x20
    @ reset NAND
    mov r1, #NAND_CTL_BASE
    ldr r2, =( (7<<12)|(7<<8)|(7<<4)|(0<<0) )
    str r2, [r1, #oNFCONF]
    ldr r2, [r1, #oNFCONF]
    ldr r2, =( (1<<4)|(0<<1)|(1<<0) )  @ Active low CE Control
    str r2, [r1, #oNFCONT]      //允许片选，允许 NAND Flash 控制寄存器
    ldr r2, [r1, #oNFCONT]

    ldr r2, =(0x6)  @ RnB Clear
```

```
    str r2, [r1, #oNFSTAT]      //清除忙标志，Rnb_TransDetect 位写“1”即可清楚
                                  忙标志
    ldr r2, [r1, #oNFSTAT]

    mov r2, #0xff   @ RESET command     //复位命令
    strb    r2, [r1, #oNFCMD]

    mov r3, #0  @ wait
nand1:
    add r3, r3, #0x1
    cmp r3, #0xa
    blt nand1

nand2:
    ldr r2, [r1, #oNFSTAT]  @ wait ready
    tst r2, #0x4
    beq nand2

    ldr r2, [r1, #oNFCONT]
    orr r2, r2, #0x2    @ Flash Memory Chip Disable
    str r2, [r1, #oNFCONT]      //关闭片选，到这里只是完成了 NAND 初始化

    @ get read to call C functions (for nand read())   //为 C 准备堆栈
    ldr sp, DW STACK START  @ setup stack pointer
    mov fp, #0  @ no previous frame, so fp=0

/*************** CHECK_NAND_FLASH_PAGE_SIZE **************************/
    ldr r2, [r1, #oNFCONF]
    tst r2, #0x08
    bne nand_page_2k
/*************** CHECK_NAND_FLASH_PAGE_SIZE ***************************/
/*************** NAND_FLASH_PAGE_SIZE_512B *****************************/
    @ copy U-Boot to RAM   r0, r1, r2 作为参数传递给函数 nand_read_ll
    ldr r0, =TEXT_BASE          //数据存放位置
    mov r1, #0x0                //复制的起始地址
    mov r2, #LENGTH_UBOOT       //复制长度
    bl  nand_read_ll            //该函数在 nand_read.c 中定义，针对 NAND Flash
                                  页大小为 512B
    tst r0, #0x0
    beq ok_nand_read

bad nand read 512:
loop2:
    b   loop2   @ infinite loop
/*************** NAND_FLASH_PAGE_SIZE_2K ******************************/
nand_page_2k:
    @ r0, r1, r2 作为参数传递给函数 nand_read_llb
    ldr r0, =TEXT_BASE          //数据存放位置
    mov r1, #0x0                //复制的起始地址
    mov r2, #LENGTH_UBOOT       //复制长度
    bl  nand_read_llb  //该函数在 nand_read.c 中定义,针对 NAND Flash 页大小为 2KB
    tst r0, #0x0
    beq ok_nand_read

bad_nand_read_2k:
loop2k:
    b   loop2k  @ infinite loop
/*************** NAND_FLASH_PAGE_SIZE_2K ******************************/
```

```
ok_nand_read:
    @ verify
    mov r0, #0
    ldr r1, =TEXT_BASE
    mov r2, #0x400  @ 4 bytes * 1024 = 4K-bytes
go_next:
    ldr r3, [r0], #4
    ldr r4, [r1], #4
    teq r3, r4
    bne notmatch    //比较前 4K 的内容是否复制正确
    subs    r2, r2, #4
    beq stack_setup
    bne go next

notmatch:
loop3:
    b   loop3   @ infinite loop
#endif

#ifdef  CONFIG_S3C2410

/* Offset */
#define oNFCONF 0x00
#define oNFCMD 0x04
#define oNFSTAT 0x10

    @ reset NAND
    mov r1, #NAND_CTL_BASE
    ldr r2, =0xf830 @ initial value
    str r2, [r1, #oNFCONF]
    ldr r2, [r1, #oNFCONF]
    bic r2, r2, #0x800  @ enable chip
    str r2, [r1, #oNFCONF]
    mov r2, #0xff       @ RESET command
    strb    r2, [r1, #oNFCMD]


    mov r3, #0  @ wait
nand1:
    add r3, r3, #0x1
    cmp r3, #0xa
    blt nand1

nand2:
    ldr r2, [r1, #oNFSTAT]  @ wait ready
    tst r2, #0x1
    beq nand2

    ldr r2, [r1, #oNFCONF]
    orr r2, r2, #0x800  @ disable chip
    str r2, [r1, #oNFCONF]

    @ get read to call C functions (for nand_read())
    ldr sp, DW STACK START  @ setup stack pointer
    mov fp, #0  @ no previous frame, so fp=0

    @ copy U-Boot to RAM
    ldr r0, =TEXT_BASE
    mov r1, #0x0
    mov r2, #LENGTH_UBOOT
```

```
    bl  nand_read_ll
    tst r0, #0x0
    beq ok nand read

bad_nand_read:
loop2:
    b   loop2   @ infinite loop


ok_nand_read:
    @ verify
    mov r0, #0
    ldr r1, =TEXT_BASE
    mov r2, #0x400  @ 4 bytes * 1024 = 4K-bytes
go_next:
    ldr r3, [r0], #4
    ldr r4, [r1], #4
    teq r3, r4
    bne notmatch
    subs    r2, r2, #4
    beq stack_setup
    bne go_next

notmatch:
loop3:
    b   loop3   @ infinite loop

#endif
```

添加对 mini2440LED1 的操作，代码的位置放在_start_armboot:.word start_armboot 之前，代码如下：

```
#if defined(CONFIG MINI2440 LED)
#define GPIO CTL BASE 0x56000000
#define oGPIO_B 0x10
#define oGPIO CON 0x0
/* R/W, Configures the pins of the port */
#define oGPIO_DAT 0x4
#define oGPIO UP 0x8
/* R/W, Pull-up disable register */
    mov r1, #GPIO CTL BASE
    add r1, r1, #oGPIO_B
    ldr r2, =0x295551
    str r2, [r1, #oGPIO CON]
    mov r2, #0xff
    str r2, [r1, #oGPIO_UP]
    ldr r2, =0x1c1
    str r2, [r1, #oGPIO_DAT]
#endif
```

在_start_armboot:.word start_armboot 之后添加堆栈地址和大小设置，代码如下：

```
#define STACK BASE 0x33f00000
#define STACK_SIZE 0x10000
    .align  2
DW_STACK_START: .word   STACK_BASE+STACK_SIZE-4
```

### 2. 添加Nand Flash读取函数

增加文件 board/samsung/mini2440/nand_read.c。添加函数 nand_read_ll 和 nand_read_llb，

分别针对 NAND Flash 页大小为 512B 和 2KB，在 include/configs/mini2440.h 中定义。

```
int
nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;

    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
    return -1;  /* invalid alignment */
    }

    NAND_CHIP_ENABLE;
    for(i=start_addr; i < (start_addr + size); i += NAND_SECTOR_SIZE) {
    /* READ0 */
        NAND_CLEAR_RB;
        NFCMD = 0;

        /* Write Address */
        NFADDR = 0;
        NFADDR = (i >> 9) & 0xff;
        NFADDR = (i >> 17) & 0xff;
        NFADDR = (i >> 25) & 0x01;

        NAND_DETECT_RB;

        for(j=0; j < NAND_SECTOR_SIZE; j++) {
            *buf = (NFDATA & 0xff);
            buf++;
            }
    }
    NAND_CHIP_DISABLE;
    return 0;
}

int
nand_read_llb(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;

    if ((start_addr & NAND_BLOCK_MASK_2K) || (size & NAND_BLOCK_MASK_2K))
{
    return -1;  /* invalid alignment */
    }

    NAND_CHIP_ENABLE;

    for(i=start_addr; i < (start_addr + size); i += NAND_SECTOR_SIZE_2K) {
    /* READ0 */
        NAND_CLEAR_RB;
        NFCMD = 0;

        /* Write Address */
        NFADDR = 0;
        NFADDR = 0;
        NFADDR = (i >> 11) & 0xff;
        NFADDR = (i >> 19) & 0xff;
        NFADDR = (i >> 27) & 0xff;

        NFCMD = 0x30;
        NAND_DETECT_RB;
```

```
        for(j=0; j < NAND_SECTOR_SIZE_2K; j++) {
            *buf = (NFDATA & 0xff);
            buf++;
            }
    }
    NAND_CHIP_DISABLE;
    return 0;
}
#endif
```

同时注意修改 board/samsung/mini2440/Makefile 文件，使得在编译 u-boot 时包含对 nand_read.c 文件的编译，修改如下：

```
COBJS   := mini2440.o nand_read.o flash.o
```

### 3．修改drivers/mtd/nand/s3c2410_nand.c文件

```
#if defined(CONFIG_S3C2410)
#endif

#if defined(CONFIG_S3C2440)
#define S3C2410_ADDR_NALE 0x08
#define S3C2410_ADDR_NCLE 0x0c

#define NFCONF          __REGi(NF_BASE + 0x0)
#define   NFCONT        __REGb(NF_BASE + 0x4)
#define NFCMD           __REGb(NF_BASE + 0x8)
#define NFADDR          __REGb(NF_BASE + 0xc)
#define NFDATA          __REGb(NF_BASE + 0x10)
#define   NFMECCD0      __REGb(NF_BASE + 0x14)
#define   NFMECCD1      __REGb(NF_BASE + 0x18)
#define   NFSECCD       __REGb(NF_BASE + 0x1c)
#define NFSTAT          __REGb(NF_BASE + 0x20)
#define NFESTAT0        __REGb(NF_BASE + 0x24)
#define NFESTAT1        __REGb(NF_BASE + 0x28)
#define   NFMECC0       __REGb(NF_BASE + 0x2c)
#define NFMECC1         __REGb(NF_BASE + 0x30)
#define NFSECC          __REGb(NF_BASE + 0x34)
#define NFSBLK          __REGb(NF_BASE + 0x38)
#define NFEBLK          __REGb(NF_BASE + 0x3c)

#define NFECC0          __REGb(NF_BASE + 0x2c)
#define NFECC1          __REGb(NF_BASE + 0x2d)
#define NFECC2          __REGb(NF_BASE + 0x2e)

#define S3C2410_NFCONT_EN          (1<<0)
#define S3C2410_NFCONT_INITECC      (1<<4)
#define S3C2410_NFCONT_nFCE        (1<<1)
#define S3C2410_NFCONT_MAINECCLOCK (1<<5)

#define S3C2410_NFCONF_TACLS(x)     ((x)<<12)
#define S3C2410_NFCONF_TWRPH0(x)    ((x)<<8)
#define S3C2410_NFCONF_TWRPH1(x)    ((x)<<4)

#endif

ulong IO_ADDR_W = NF_BASE;
```

将所有

```
#if defined(CONFIG_S3C2410)
```

改为

```
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
```

### 3.3.4　具体平台相关修改

该部分主要修改与具体开发板相关的内容，该部分修改的文件在 mini2440 子目录下，修改文件包括 lowlevel_init.S、mini2440.c、s3c2410_nand.c 等。

#### 1．修改board/samsung/mini2440/lowlevel_init.S文件

修改 lowlevel_init.S 时参考 mini2440 内存控制文档。

```
#if defined(CONFIG_S3C2440)
#define Trp         0x2 /* 4clk */       //SDRAM RAS 预充电时间
#define REFCNT      1012                 //SDRAM 刷新计数值
#else
#define Trp         0x0 /* 2clk */
#define REFCNT      0x0459
#endif
```

#### 2．针对GPIO和PLL的配置，修改文件board/Samsung/mini2440/mini2440.c

在 mini2440.c 文件中添加下列头文件。

```
#include <video_fb.h>
#if defined(CONFIG_CMD_NAND)
#include <linux/mtd/nand.h>
#endif
#include <net.h>
#include <netdev.h>
```

区别 2410 和 2440 不同的时钟频率。

```
#define U_M_MDIV    0x48
#define U_M_PDIV    0x3
```

为：

```
#if defined(CONFIG_S3C2410)
/* Fout = 202.8MHz */
#define M_MDIV  0xA1
#define M_PDIV  0x3
#define M_SDIV  0x1
#endif
#if defined(CONFIG_S3C2440)
/* Fout = 405MHz */
#define M_MDIV 0x7f
#define M_PDIV 0x2
#define M_SDIV 0x1
#endif
#endif
```

2440 板初始化函数修改如下：

```
int board_init (void)
{
    S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
    S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();
    /* to reduce PLL lock time, adjust the LOCKTIME register */
    clk_power->LOCKTIME = 0xFFFFFF;
    /* configure MPLL */
    clk_power->MPLLCON = ((M_MDIV << 12) + (M_PDIV << 4) + M_SDIV);
    /* some delay between MPLL and UPLL */
    delay (4000);
    /* configure UPLL */
    clk_power->UPLLCON = ((U_M_MDIV << 12) + (U_M_PDIV << 4) + U_M_SDIV);
    /* some delay between MPLL and UPLL */
    delay (8000);
    /* set up the I/O ports */
    gpio->GPACON = 0x007FFFFF;
#if defined(CONFIG_MINI2440)
    gpio->GPBCON = 0x00295551;
#else
    gpio->GPBCON = 0x00044556;
#endif
    gpio->GPBUP = 0x000007FF;
    gpio->GPCCON = 0xAAAAAAAA;
    gpio->GPCUP = 0xFFFFFFFF;
    gpio->GPDCON = 0xAAAAAAAA;
    gpio->GPDUP = 0xFFFFFFFF;
   gpio->GPECON = 0xAAAAAAAA;
    gpio->GPEUP = 0x0000FFFF;
    gpio->GPFCON = 0x000055AA;
    gpio->GPFUP = 0x000000FF;
    gpio->GPGCON = 0xFF95FF3A;
    gpio->GPGUP = 0x0000FFFF;
    gpio->GPHCON = 0x0016FAAA;
    gpio->GPHUP = 0x000007FF;
    gpio->EXTINT0=0x22222222;
    gpio->EXTINT1=0x22222222;
    gpio->EXTINT2=0x22222222;
#if defined(CONFIG_S3C2410)
    /* arch number of SMDK2410-Board */
    gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
#endif

#if defined(CONFIG_S3C2440)
/* arch number of S3C2440-Board */
    gd->bd->bi_arch_number = MACH_TYPE_S3C2440 ;
#endif

    /* adress of boot parameters */
    gd->bd->bi_boot_params = 0x30000100;

    icache_enable();
    dcache_enable();
#if defined(CONFIG_MINI2440_LED)
    gpio->GPBDAT = 0x00000181;
#endif
    return 0;
}
```

## 3.3.5　其他部分修改

该部分包括一些头文件的修改、部分驱动的修改、时钟计算方法修改等。

### 1．对于drivers/rtc/s3c24x0_rtc.c文件的修改

```
#elif defined(CONFIG_S3C2410)
```

修改为：

```
#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
```

### 2．需要在include/s3c24x0.h文件中添加CAMDIVN定义

```
typedef struct {
    S3C24X0_REG32   LOCKTIME;
    S3C24X0_REG32   MPLLCON;
    S3C24X0_REG32   UPLLCON;
    S3C24X0_REG32   CLKCON;
    S3C24X0_REG32   CLKSLOW;
    S3C24X0_REG32   CLKDIVN;
#if defined (CONFIG_S3C2440)
    S3C24X0_REG32  CAMDIVN;
#endif
} /*__attribute__((__packed__))*/ S3C24X0_CLOCK_POWER;
```

将所有

```
#if defined(CONFIG_S3C2410)
```

改为

```
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
```

根据 2440 芯片资料 6.3 节 NAND Flash 寄存器，修改寄存器定义。

```
#if defined(CONFIG_S3C2410)
/* NAND FLASH (see S3C2410 manual chapter 6) */
typedef struct {
    S3C24X0_REG32   NFCONF;
    S3C24X0_REG32   NFCMD;
    S3C24X0_REG32   NFADDR;
    S3C24X0_REG32   NFDATA;
    S3C24X0_REG32   NFSTAT;
    S3C24X0_REG32   NFECC;
} /*__attribute__((__packed__))*/ S3C2410_NAND;
#endif
#if defined (CONFIG_S3C2440)
/* NAND FLASH (see S3C2440 manual chapter 6) */
typedef struct {
    S3C24X0_REG32 NFCONF;       //NAND Flash 配置寄存器
    S3C24X0_REG32 NFCONT;       //NAND Flash 控制寄存器
    S3C24X0_REG32 NFCMD;        //NAND Flash 命令寄存器
    S3C24X0_REG32 NFADDR;       //NAND Flash 地址寄存器
    S3C24X0_REG32 NFDATA;       //NAND Flash 数据寄存器
    S3C24X0_REG32   NFMECCD0;   //NAND Flash 主数据区域 ECC 寄存器 0
```

```
    S3C24X0_REG32   NFMECCD1;    //NAND Flash 主数据区域 ECC 寄存器 1
    S3C24X0_REG32   NFSECCD;     //NAND Flash 空闲区域 ECC 寄存器
    S3C24X0_REG32   NFSTAT;      //NAND Flash 操作状态寄存器
    S3C24X0_REG32   NFESTAT0;    //NAND FlashECC0 状态寄存器
    S3C24X0_REG32   NFESTAT1;    //NAND FlashECC1 状态寄存器
    S3C24X0_REG32   NFMECC0;     //NAND Flash 主数据区域 ECC0 状态寄存器
    S3C24X0_REG32   NFMECC1;     //NAND Flash 主数据区域 ECC1 状态寄存器
    S3C24X0_REG32   NFSECC;      //NAND Flash 空闲区域 ECC 状态寄存器
    S3C24X0_REG32   NFSBLK;      //NAND Flash 块地址寄存器
    S3C24X0_REG32   NFEBLK;      //NAND Flash 块地址寄存器
} /*__attribute__((__packed__))*/ S3C2410_NAND;
#endif
```

在结构/*__attribute__((__packed__))*/ S3C24X0_GPIO 中添加 2440 数据项。

```
#if defined (CONFIG_S3C2440)
    S3C24X0_REG32   res9[3];
    S3C24X0_REG32   MSLCON;
    S3C24X0_REG32   GPJCON;
    S3C24X0_REG32   GPJDAT;
    S3C24X0_REG32   GPJUP;
#endif
```

### 3．修改include目录下common.h和serial.h， cpu/arm920t/s3c24x0目录下interrupts.c和usb.c文件，以及drivers/serial/serial_s3c24x0.c文件

将所有

```
#if defined(CONFIG_S3C2410)
```

改为

```
#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
```

### 4．修改cpu/arm920t/s3c24x0/speed.h中的函数get_PLLCLK()和get_HCLK()

2410 和 2440 的时钟计算方法不相同，依据不同类型的板计算其时钟。

```
static ulong get_PLLCLK(int pllreg)
{
   S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
   ulong r, m, p, s;

   if (pllreg == MPLL)
    r = clk_power->MPLLCON;
   else if (pllreg == UPLL)
    r = clk_power->UPLLCON;
   else
    hang();
   m = ((r & 0xFF000) >> 12) + 8;
   p = ((r & 0x003F0) >> 4) + 2;
   s = r & 0x3;
#if defined(CONFIG_S3C2440)
    if (pllreg == MPLL)
    return ((CONFIG_SYS_CLK_FREQ * m * 2) /(p << s));
    else if (pllreg == UPLL)
#endif
   return((CONFIG_SYS_CLK_FREQ * m) / (p << s));
```

```
}
ulong get_HCLK(void)
{
    S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
#if defined(CONFIG_S3C2440)
    if (clk_power->CLKDIVN & 0x6)
                {
                if ((clk_power->CLKDIVN & 0x6)==2) return(get_FCLK()/2);
                if ((clk_power->CLKDIVN & 0x6)==6) return((clk_power->
                CAMDIVN & 0x100) ? get_FCLK()/6 : get_FCLK()/3);
                if ((clk_power->CLKDIVN & 0x6)==4) return((clk_power->
                CAMDIVN & 0x200) ? get_FCLK()/8 : get_FCLK()/4);
                return(get_FCLK());
                }
    else    return(get_FCLK());
#else
    return((clk_power->CLKDIVN & 0x2) ? get_FCLK()/2 : get_FCLK());
#endif
}
```

### 5．在include/configs/mini2440.h头文件中添加宏定义

删除以下内容：

```
#define CONFIG_SMDK2410         1  /* on a SAMSUNG SMDK2410 Board  */
#define CONFIG_S3C2410  1
```

添加对 2440 的宏定义。

```
#define CONFIG_S3C2440          1 /* in a SAMSUNG S3C2440 SoC */
#define CONFIG_MINI2440         1 /* on a SAMSUNG mini2440 Board */
#define CONFIG_MINI2440_LED     1 /* Use the LED on Board */
```

并增加以下内容：

```
#define CONFIG_CMD_NAND  /* NAND support   */
#define CFG_ENV_IS_IN_NAND  1    /*  */
#define CFG_ENV_OFFSET   0X40000
/*
u-boot:0x00000--0x40000,param:0x40000--0x60000,kernel:0x60000--0x260000
128K block*/
#define CFG_ENV_SIZE  0x10000 /* Total Size of Environment Sector */
//#define CONFIG_SETUP_MEMORY_TAGS 1
//#define CONFIG_CMDLINE_TAG 1
/*nand flash define*/
#if defined(CONFIG_CMD_NAND)
#define CFG_MAX_NAND_DEVICE 1          //just one nand flash chip on board
#define CFG_NAND_BASE      0x4e000000   //nand flash base address
#define SECTORSIZE 2048
/* one page size*/
#define NAND_SECTOR_SIZE SECTORSIZE
#define NAND_BLOCK_MASK (SECTORSIZE - 1)
/* Nandflash Boot */
#define CONFIG_S3C2440_NAND_BOOT 1
#endif
```

修改命名行字符串

```
#define CONFIG_SYS_PROMPT       "SMDK2410 # "
```

为：

```
#define CONFIG_SYS_PROMPT        "MINI2440 # "
```

修改默认的载入地址

```
#define CONFIG_SYS_LOAD_ADDR        0x33000000
```

为：

```
#define CONFIG_SYS_LOAD_ADDR        0x30008000
```

## 3.3.6　U-Boot 的编译

该部分主要介绍在编译过程中会遇到的一些常见的错误，同时给出错误出现的原因，减少读者编译的时间；另外，给出编译执行的命令。

### 1．编译可能会遇到的错误

当遇到编译器的问题时，可以查看 config.mk 文件，例如浮点数错误：

```
arm-linux-ld: ERROR: /usr/local/arm/3.4.1/lib/gcc/arm-linux/3.4.1/libgcc.
a(_udivdi3.oS) uses hardware FP, whereas u-boot uses software FP
```

将 cpu/arm920t/config.mk 文件中的

```
PLATFORM_RELFLAGS += -fno-strict-aliasing  -fno-common -ffixed-r8 \
    -msoft-float
```

改为

```
PLATFORM_RELFLAGS += -fno-strict-aliasing  -fno-common -ffixed-r8
arm-linux-ld: ERROR: Source object
```

当使用 4.3.2 时可能会遇到下面的错误：

```
/usr/local/arm/4.3.2/bin/../lib/gcc/arm-none-linux-gnueabi/4.3.2/armv4t
/libgcc.a( udivdi3.o) has EABI version 5, but target u-boot has EABI version
0
arm-linux-ld:   failed   to   merge   target   specific   data   of   file
/usr/local/arm/4.3.2/bin/../lib/gcc/arm-none-linux-gnueabi/4.3.2/armv4t
/libgcc.a(_udivdi3.o)
```

是通过修改 cpu/arm920t/config.mk 的内容。把最后一行

```
PLATFORM_RELFLAGS +=$(call cc-option,-mshort-load-bytes,$(call cc-option,
-malignment-traps,))
```

修改成

```
PLATFORM_RELFLAGS +=$(call  cc-option,$(call  cc-option,))
```

### 2．编译生成u-boot.bin文件

```
#make clean
#make mini2440_config
#make
```

# 3.4　Bootloader 之 vivi

vivi 属于当前比较流行的，专门针对 ARM9 处理器而设计的一款 Bootloader，其特点是操作简便，同时具有完备的命令体系。因此，对其分析和研究 vivi 有利于将其移植到目前流行的 ARM9 处理器上。虽然其功能不如 u-boot 强大，但是其移植过程相对简单。本节主要介绍主要代码和移植方法。

## 3.4.1　vivi 简介

vivi 是韩国 mizi 公司开发的 bootloader，适用于 ARM9 处理器。 与大多数 Bootloader 类似，vivi 也具有两种工作模式：“启动加载”模式和“下载”模式。启动加载模式可以在一段时间后（这个时间可更改）自动启动 Linux 内核，这是 vivi 的正常的工作模式。在下载模式下，vivi 为用户提供一个命令行接口，通过这一接口用户可以通过终端向开发板输入命令来操控开发板，vivi 的主要命令见表 3.2。

表 3.2　vivi命令

| 命　　令 | 功　　能 |
|---|---|
| load | 把二进制文件载入 Flash 或 RAM |
| part | 操作 MTD 分区信息。显示、增加、删除、复位、保存 MTD 分区 |
| parm | 设置参数 |
| boot | 启动系统 |
| flash | 管理 Flash，如删除 Flash 的数据 |

## 3.4.2　vivi 配置与编译

进入 vivi 目录，使用 VI 工具打开 Makefile，找到交叉编译器、内核头文件路径、编译器库文件路径等。vivi-0.1.4 版本的默认设置为：

```
CROSS_COMPILE   = arm-linux-
LINUX_INCLUDE_DIR       =
LIBC_INCLUDE_DIR       = $(CROSSDIR)/arm-linux/include
ARM_GCC_LIBS    = /usr/lib/gcc-lib/arm-linux/2.95.3
ARM_C_LIBS      = /usr/arm-linux/lib
```

本机使用的内核为 linux-2.6.29，编译器使用的版本为 4.3.2。找到本机上内核头文件路径、编译器库文件路径等，替换旧路径。Makefile 的主要需要修改部分如下：

```
ARCH := arm
LINUX_INCLUDE_DIR   =
/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/usr/include
CROSS_COMPILE   = /usr/local/arm/4.3.2/bin/arm-linux-
#normal flags
CFLAGS := $(CPPFLAGS) -Wall -Wstrict-prototypes -O2 -fPIC -fomit-frame-
pointer
```

```
ARM_GCC_LIBS= /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/armv4t/lib
CLIBS = -L$(ARM_GCC_LIBS) -L/usr/local/arm/4.3.2/arm-none-linux-gnueabi/
libc/armv4t/usr/lib -L/usr/local/arm/4.3.2/lib/gcc/arm-none-linux-gnueabi/
4.3.2/armv4t/ -lgcc -lc
```

使用 make distclean 在配置前进行清理，使用 make menuconfig 进行配置。在配置 vivi 时，通过选择 System Type --->，再选择 () ARM system type，最后选择(X) S3C2440-based，如图 3.2 所示。




图 3.2　vivi 的 System Type 选择

在 Implementation 的 Platform 选项中选择支持 NAND 启动，因为 mini2440 只有 Nand Flash 没有 Nor Flash，所以要选择 NAND Boot 启动，如图 3.3 所示。




图 3.3　选择支持 NAND 启动

对于 General setup 选项的配置，支持 I-Cache 和 D-Cache，如图 3.4 所示。




图 3.4　对于 Cache 的配置

另外，还要配置串口，通过串口来调试和输出信息，串口的配置如图 3.5 所示。




图 3.5　串口的配置

对于 vivi 命令的配置，需要配置 mem command（内存操作命令）、param command（参数设置命令）、part command（设置分区信息命令）和 bon command（分区命令），如图 3.6 所示。最后退出并保存。




图 3.6　配置 vivi 命令

使用 make 命令进行编译，在 vivi 目录下生成名字为 vivi 的文件，通过 JTAG 烧写到板子上。

## 3.4.3　代码分析

vivi 的代码包括 arch、init、lib、drivers 和 include 等几个目录。下面分别对这几个目录的功能进行介绍。

- arch：此目录包括了所有 vivi 支持的目标板子的子目录，例如 s3c3410、s3c2440 目录。
- drivers：其中包括了引导内核需要的设备的驱动程序（MTD、net 和串口）。MTD 目录下分为 map、nand 和 nor 三个目录。
- init：此目录下只存在 main.c 和 version.c 两个文件。vivi 和普通的 C 程序一样，从 main()函数开始执行。
- lib：此目录包括一些平台公共的接口代码，比如 time.c 里的 udelay()和 medlay()，memory.c 里的 command_mem_copy()和 command_mem_info()。
- include：是头文件的公共目录，其中的 s3c2440.h 定义了这块处理器的一些寄存器。platform/smdk2440.h 定义了与开发板相关的资源配置参数，处理器类型、CPU 时钟等。

# 3.5　vivi 的运行

vivi 的运行分为两个阶段，阶段 1 的代码在 arch/s3c2440 0/head.S 中，阶段 2 的代码从

init/main.c 的 main()函数开始。

### 3.5.1　Bootloader 启动的阶段一

该阶段为 Bootloader 的第一个阶段，该阶段完成的内容包括，关闭看门狗、禁止所有中断、初始化系统时钟、初始化控制器、点亮 LED、设置 GPIO，将所有的代码从 NandFlash 复制到 SRAM 中，为 Bootloader 的下一个阶段做准备。

```
@ Start VIVI head
Reset:
    @上电后，WATCH DOG 默认是开着的，关闭 WATCH DOG
    mov r1, #0x53000000
    mov r2, #0x0
    str r2, [r1]
    @禁止所有中断，vivi 中不会用到中断，中断是系统的事
    mov r1, #INT_CTL_BASE
    mov r2, #0xffffffff
    str r2, [r1, #oINTMSK]
    ldr r2, =0x7ff
    str r2, [r1, #oINTSUBMSK]
    @初始化系统时钟
    mov r1, #CLK_CTL_BASE
    mvn r2, #0xff000000
    str r2, [r1, #oLOCKTIME]

    mov r1, #CLK_CTL_BASE
    ldr r2, clkdivn_value
    str r2, [r1, #oCLKDIVN]

    mrc p15, 0, r1, c1, c0, 0       @ read ctrl register
    orr r1, r1, #0xc0000000         @ Asynchronous
    mcr p15, 0, r1, c1, c0, 0       @ write ctrl register

    mov r1, #CLK_CTL_BASE
    @ldr    r2, mpll_value          @ clock default
    ldr r2, =0x7f021    @mpll_value_USER        @ clock user set
    str r2, [r1, #oMPLLCON]
    bl  memsetup                            @跳转到 memsetup 函数
#ifdef CONFIG_PM    vivi 考虑不需要使用电源管理
    @检查是否从掉电模式唤醒，若是则调用 WakeupStart 函数进行处理
    ldr r1, PMST_ADDR
    ldr r0, [r1]
    tst r0, #(PMST_SMR)
    bne WakeupStart      @查看状态，判断是否需要跳转到 WakeupStart
#endif
    @ All LED on 点亮开发板上所有 LED
    mov r1, #GPIO_CTL_BASE
    add r1, r1, #oGPIO_F  @LED 使用 GPIOF 组的管脚
    ldr r2,=0x55aa  @使能 EINT0，ENIT1，EINT2，EINT3，另四个管脚配置成输出，屏蔽
    EINT4，5，6，7
    str r2, [r1, #oGPIO_CON]
    mov r2, #0xff
    str r2, [r1, #oGPIO_UP]
    mov r2, #0x00
    str r2, [r1, #oGPIO_DAT]
```

```
#if 0
    @ SVC
    mrs r0, cpsr
    bic r0, r0, #0xdf
    orr r1, r0, #0xd3
    msr cpsr_all, r1
#endif

    @ set GPIO for UART   设置串口
#ifdef CONFIG_S3C2440_SMDK
    mov r1, #GPIO_CTL_BASE
    add r1, r1, #oGPIO_H      @设置 GPIO_H 组管脚为串口
    ldr r2, gpio_con_uart
    str r2, [r1, #oGPIO_CON]
    ldr r2, gpio_up_uart
    str r2, [r1, #oGPIO_UP]
#endif
    bl  InitUART

#ifdef CONFIG_DEBUG_LL
    @ Print current Program Counter
    ldr r1, SerBase
    mov r0, #'\r'
    bl  PrintChar
    mov r0, #'\n'
    bl  PrintChar
    mov r0, #'@'
    bl  PrintChar
    mov r0, pc
    bl  PrintHexWord
#endif

#ifdef CONFIG_S3C2440_NAND_BOOT  @如果定义了以 Nand flash 方式启动
    bl  copy_myself  @从 Nand flash 复制到 SDRAM 中

#if 1
    mov r1, #GPIO_CTL_BASE  @nand flash 的地址
    add r1, r1, #oGPIO_F
    mov r2, #0x00
    str r2, [r1, #oGPIO_DAT]  @复制的长度
#endif

    @ jump to ram
    ldr r1, =on_the_ram
    add pc, r1, #0
    nop
    nop
1:  b   1b      @ infinite loop

on_the_ram:
#endif

#ifdef CONFIG_DEBUG_LL      @打印调试信息
    ldr r1, SerBase
    ldr r0, STR_STACK
    bl  PrintWord
    ldr r0, DW_STACK_START  @设置堆栈的地址，nand_read_ll 的参数
    bl  PrintHexWord
#endif
```

```
    @跳到 bootloader 的阶段 2 运行，亦即调用 init/main.c 中的 main()函数
    ldr sp, DW STACK START  @ setup stack pointer
    mov fp, #0              @ no previous frame, so fp=0
    mov a2, #0              @ set argv to NULL

    bl  main                @ 调用主函数

    mov pc, #FLASH BASE     @ otherwise, reboot
@ End VIVI head

@ Wake-up codes
#ifdef CONFIG_PM
WakeupStart:
    @ Clear sleep reset bit
    ldr r0, PMST_ADDR
    mov r1, #PMST_SMR
    str r1, [r0]

    @ Release the SDRAM signal protections
    ldr r0, PMCTL1 ADDR
    ldr r1, [r0]
    bic r1, r1, #(SCLKE | SCLK1 | SCLK0)
    str r1, [r0]

    @ Go...
    ldr r0, PMSR0 ADDR  @ read a return address
    ldr r1, [r0]
    mov pc, r1
    nop
    nop
1:  b   1b      @ infinite loop

SleepRamProc:
    @ SDRAM is in the self-refresh mode */
    ldr r0, REFR_ADDR
    ldr r1, [r0]
    orr r1, r1, #SELF REFRESH
    str r1, [r0]

    @ wait until SDRAM into self-refresh
    mov r1, #16
1:  subs    r1, r1, #1
    bne 1b

    @ Set the SDRAM singal protections
    ldr r0, PMCTL1 ADDR
    ldr r1, [r0]
    orr r1, r1, #(SCLKE | SCLK1 | SCLK0)
    str r1, [r0]

    /* Sleep... Now */
    ldr r0, PMCTL0 ADDR
    ldr r1, [r0]
    orr r1, r1, #SLEEP ON
    str r1, [r0]
1:  b   1b

#ifdef CONFIG_TEST
hmi:
    ldr r0, PMCTL0 ADDR
    ldr r1, =0x7fff0
```

```
    str r1, [r0]

    @ All LED on
    mov r1, #GPIO_CTL_BASE
    add r1, r1, #oGPIO_F
    ldr r2,=0x55aa
    str r2, [r1, #oGPIO_CON]
    mov r2, #0xff
    str r2, [r1, #oGPIO_UP]
    mov r2, #0xe0
    str r2, [r1, #oGPIO_DAT]
1:  b   1b
#endif

#endif

ENTRY(memsetup)                          @memsetup 入口
    @ 初始化静态内存
    @ set memory control registers
    mov r1, #MEM_CTL_BASE
    adrl    r2, mem_cfg_val
    add r3, r1, #52
1:  ldr r4, [r2], #4
    str r4, [r1], #4
    cmp r1, r3
    bne 1b
    mov pc, lr

#ifdef CONFIG_S3C2440_NAND_BOOT
@ copy_myself: copy vivi to ram
copy_myself:
    mov r10, lr

    @ reset NAND
    mov r1, #NAND_CTL_BASE
    ldr r2, =( (7<<12)|(7<<8)|(7<<4)|(0<<0) )
    str r2, [r1, #oNFCONF]
    ldr r2, [r1, #oNFCONF]

    ldr r2, =( (1<<4)|(0<<1)|(1<<0) ) @ Active low CE Control
    str r2, [r1, #oNFCONT]
    ldr r2, [r1, #oNFCONT]

    ldr r2, =(0x6)      @ RnB Clear
    str r2, [r1, #oNFSTAT]
    ldr r2, [r1, #oNFSTAT]

    mov r2, #0xff       @ RESET command
    strb    r2, [r1, #oNFCMD]
    mov r3, #0          @ wait
1:  add r3, r3, #0x1
    cmp r3, #0xa
    blt 1b
2:  ldr r2, [r1, #oNFSTAT]  @ wait ready
    tst r2, #0x4
    beq 2b

    ldr r2, [r1, #oNFCONT]
    orr r2, r2, #0x2        @ Flash Memory Chip Disable
    str r2, [r1, #oNFCONT]
```

```
    @ get read to call C functions (for nand_read())
    ldr sp, DW_STACK_START  @ setup stack pointer
    mov fp, #0           @ no previous frame, so fp=0

    mov r1, #GPIO_CTL_BASE
    add r1, r1, #oGPIO_F
    mov r2, #0xe0
    str r2, [r1, #oGPIO_DAT]

    @ copy vivi to RAM
    ldr r0, =VIVI_RAM_BASE
    mov     r1, #0x0
    mov r2, #0x20000     @128k 字节
    bl  nand_read_ll   @跳转到 nand_read_ll 函数，r0，r1，r2 分别为三个参数，从
    NANDFlash 的 0 地址复制 128k 到 SDRAM 指定处

#if 1
    mov r1, #GPIO_CTL_BASE
    add r1, r1, #oGPIO_F
    mov r2, #0xb0
    str r2, [r1, #oGPIO_DAT]
#endif

    tst r0, #0x0
    beq ok_nand_read
#ifdef CONFIG_DEBUG_LL
bad_nand_read:
    ldr r0, STR_FAIL
    ldr r1, SerBase
    bl  PrintWord
1:  b   1b      @ infinite loop
#endif

ok_nand_read:
#ifdef CONFIG_DEBUG_LL
    ldr r0, STR_OK
    ldr r1, SerBase
    bl  PrintWord
#endif
    @ verify

    mov r0, #0
    ldr r1, =0x33f00000
    mov r2, #0x400  @ 4 bytes * 1024 = 4K-bytes
go_next:
    ldr r3, [r0], #4
    ldr r4, [r1], #4
    teq r3, r4
    bne notmatch
    subs    r2, r2, #4
    beq done_nand_read
    bne go_next
notmatch:
#ifdef CONFIG_DEBUG_LL
    sub r0, r0, #4
    ldr r1, SerBase
    bl  PrintHexWord
    ldr r0, STR_FAIL
    ldr r1, SerBase
    bl  PrintWord
#endif
```

```
1:  b   1b
done_nand_read:

#ifdef CONFIG_DEBUG_LL
    ldr r0, STR_OK
    ldr r1, SerBase
    bl  PrintWord
#endif

#if 1
    mov r1, #GPIO_CTL_BASE
    add r1, r1, #oGPIO_F
    mov r2, #0x70
    str r2, [r1, #oGPIO_DAT]
#endif
    mov pc, r10  @vivi 复制到 SDRAM 完成，函数返回

@ clear memory   执行清理工作
@ r0: start address
@ r1: length
mem_clear:
    mov r2, #0
    mov r3, r2
    mov r4, r2
    mov r5, r2
    mov r6, r2
    mov r7, r2
    mov r8, r2
    mov r9, r2

clear_loop:
    stmia   r0!, {r2-r9}
    subs    r1, r1, #(8 * 4)
    bne clear_loop

    mov pc, lr

#endif @ CONFIG_S3C2440_NAND_BOOT
@ Initialize UART
@
@ r0 = number of UART port
InitUART:
    ldr r1, SerBase
    mov r2, #0x0
    str r2, [r1, #oUFCON]
    str r2, [r1, #oUMCON]
    mov r2, #0x3
    str r2, [r1, #oULCON]
    ldr r2, =0x245
    str r2, [r1, #oUCON]
#define UART_BRD ((UART_PCLK  / (UART_BAUD_RATE * 16)) - 1)
    mov r2, #UART_BRD
    str r2, [r1, #oUBRDIV]

    mov r3, #100
    mov r2, #0x0
1:  sub r3, r3, #0x1
    tst r2, r3
    bne 1b
```

### 3.5.2　Bootloader 启动的阶段二

vivi 的第二阶段是从 main()函数开始，该函数总共可以分为 8 个步骤。

```
int main(int argc, char *argv[])
{
    int ret;
    GPFDAT = 0x10;
    /* NB: MMU off state */
    /*第一步打印 vivi 版本*/
    putstr("\r\n");
    putstr(vivi_banner);
    reset_handler();
    /*第二步：对开发板进行初始化*/
    ret = board_init();
    GPFDAT = 0x20;
    if (ret) {
        putstr("Failed a board init() procedure\r\n");
        error();
    }
    /*第三步：内存映射初始化和内存管理单元的初始化工作*/
    mem_map_init();
    mmu_init();
    putstr("Succeed memory mapping.\r\n");
    /*第四步：调用了 heap_init(void)函数，并返回值。该值是函数 heap_init()调用的
    mmalloc_init()函数的返回值，申请一块内存区域*/
    /* initialize the heap area*/
    ret = heap_init();          //vivi/lib/heap.c 中定义
    if (ret) {
        putstr("Failed initailizing heap region\r\n");
        error();
    }
    /*第五步：MTD 设备的初始化*/
    ret = mtd_dev_init();
    /*第六步：读取 bootloader 参数*/
    init_priv_data();           //vivi/lib/priv_data/rw.c 中定义
    /*第七步：初始化内置命令*/
    misc();
    init_builtin_cmds();
    /*第八步：启动 boot_or_vivi()。启动成功后，将通过 vivi_shell()启动一个 shell
    （如果配置了 CONFIG_SERIAL_TERM），此时 vivi 的任务完成*/
    boot_or_vivi();
    return 0;
}
```

## 3.6　小　　结

本章主要介绍了 Bootloader 的启动流程，以及这个启动流程在 U-Boot 和 vivi 上是如何体现的。本章还介绍了两种 Bootloader 在 ARM 平台的移植，通过介绍移植主要步骤，使读者大致了解移植 Bootloader 时需要注意的方面；通过代码分析让读者清楚 Bootloader 执行流程和功能。

# 第 4 章　Linux 内核裁剪与移植

内核，即操作系统。它为底层的可编程部件提供服务，为上层应用程序提供执行环境。内核裁剪就是对这些功能进行裁剪，选取满足特定平台和需求的功能。不同的硬件平台对内核要求也不同，因此从一个平台到另一个平台需要对内核进行重新配置和编译。操作系统从一个平台过渡到另一个平台称为移植。Linux 是一款平台适应性且容易裁剪的操作系统，因此 Linux 在嵌入式系统得到了广泛的应用。本章将详细讲解内核裁剪与移植的各项技术。

## 4.1　Linux 内核结构

Linux 内核采用模块化设计，并且各个模块源码以文件目录的形式存放，在对内核的裁剪和编译时非常方便。下面介绍内核的主要部分及其文件目录。

### 4.1.1　内核的主要组成部分

在第 1 章中已经介绍了 Linux 内核主要的 5 个部分：进程调度、内存管理、虚拟文件系统、网络接口、进程通信。在系统移植的时候，它们是内核的基本元素，这 5 个部分之间的关系，如图 4.1 所示。

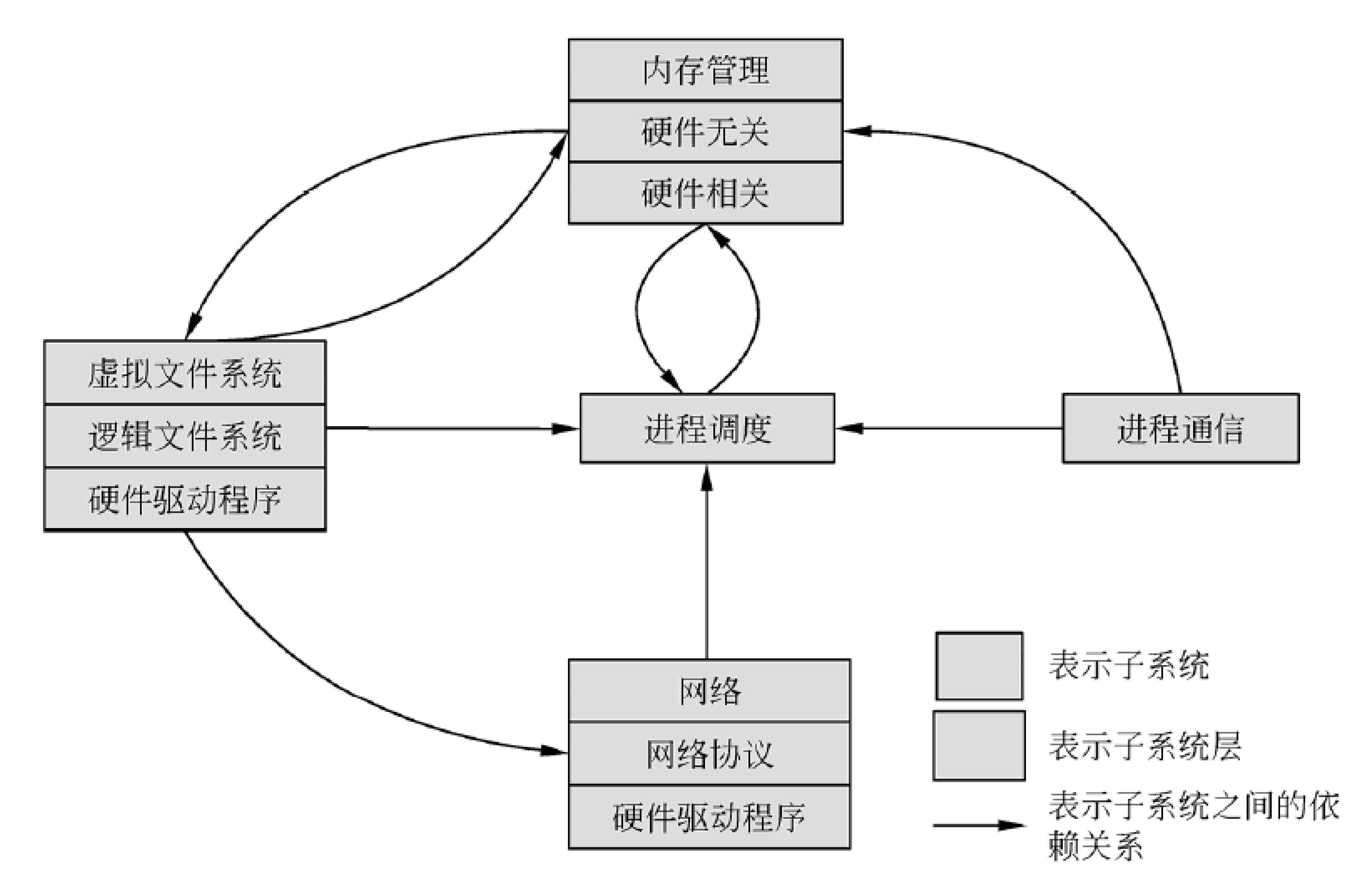



图 4.1　Linux 内核子系统及其之间的关系

进程调度部分负责控制进程对 CPU 的访问。内存管理允许多个进程安全地共享主内存区域。内存管理从逻辑上分为硬件无关部分和硬件相关部分。硬件无关部分提供了进程的映射和逻辑内存的对换；硬件相关部分为内存管理硬件提供了虚拟接口。虚拟文件系统隐藏了不同类型硬件的具体细节，为所有的硬件设备提供了一个标准的接口，VFS 提供了十多种不同类型的文件系统。网络接口提供了对各种网络标准的存取和各种网络硬件的支持。进程通信部分用于支持进程间各种不同的通信机制。进程调度处于核心位置，内核的其他子系统都要依赖它，因为每个子系统都存在进程挂起或恢复过程。

- 进程调度与内存管理之间的关系：这两个子系统为互相依赖关系。在多道程序环境下，程序要运行必须为之创建进程，而创建进程首先就是要将程序和数据装入内存。另外，内存管理子系统也存在进程的挂起和恢复过程。
- 进程间通信与内存管理之间的关系：进程间通信子系统要依赖内存管理支持共享内存通信机制，通过对共同的内存区域进行操作来达到通信的目的。
- 虚拟文件系统与网络接口之间的关系：虚拟文件系统通过依赖网络接口支持网络文件系统（NFS），也通过依赖内存管理支持 RAMDISK 设备。
- 内存管理与虚拟文件系统之间的关系：内存管理利用虚拟文件系统支持交换，交换进程定期地由调度程序调度，这也是内存管理依赖于进程调度的唯一原因。当一个进程存取的内存映射被换出时，内存管理将会向文件系统发出请求，同时，挂起当前正在运行的进程。

除了上面 5 个主要部分，下面将介绍 Linux 代码的整体分区结构。

## 4.1.2　内核源码目录介绍

Linux 内核代码以源码树的形式存放，如果在安装系统的时候已经安装了源码树，其源码树就在/usr/src/linux 下，源码树结构如图 4.2 所示。

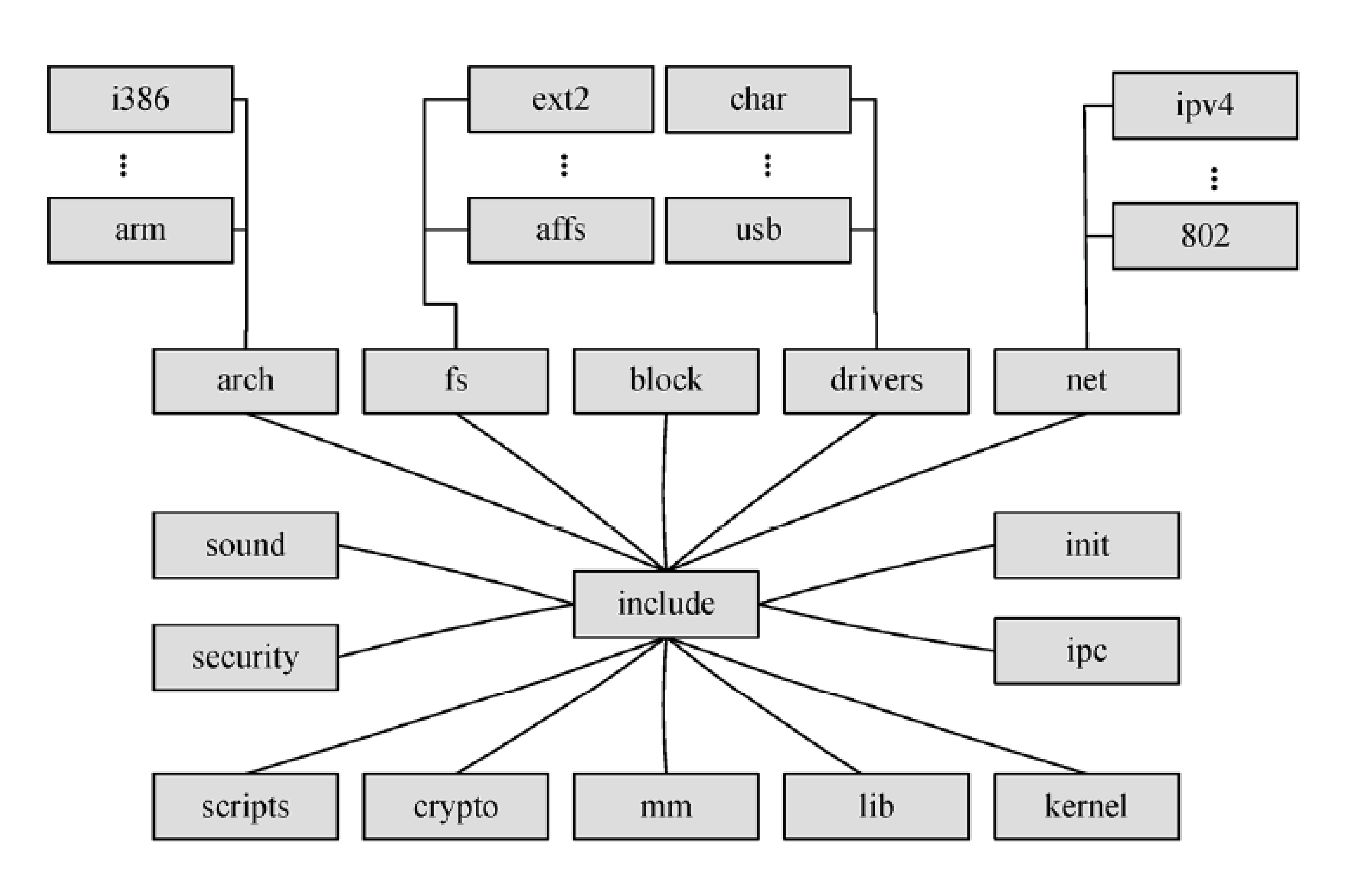



图 4.2　Linux 内核源码树结构

下面分别针对图 4.2 中各个部分进行介绍，各个目录的主要的功能分别如下。

### 1．arch目录

arch 子目录包括了所有和体系结构相关的核心代码。它的每一个子目录都代表一种支持的体系结构，例如 arm 子目录是关于 ARM 平台下各种芯片兼容的代码。

### 2．include目录

include 子目录包括内核编译时所需要的大部分头文件。与平台无关的头文件在 include/linux 子目录下，include/scsi 目录则是有关 scsi 设备的头文件目录，与 arm 相关的头文件在 include/asm-arm 子目录下。

### 3．drivers目录

drivers 子目录放置系统所有的设备驱动程序。有些驱动是与硬件无关的，而有些驱动是与硬件平台相关。例如，在 USB 驱动中，主机控制器有 3 种规格：

- OHCI 主要为非 PC 系统上及带有 SiShe ALi 芯片组的 PC 主板上的 USB 芯片，嵌入式系统一般使用该驱动。
- UHCI 大多为 Intel 和 Via 主板上的 USB 控制器芯片。相对 OHCI 而言 UHCI 的硬件电路比较简单，同时其成本也比较低，但驱动复杂，但它们都是在 USB 1.1 规范同时提出的。
- EHCI 由 USB 2.0 规范所提出，它兼容 OHCI 和 UHCI。

### 4．fs目录

fs 子目录列出了 Linux 支持的所有文件系统，目前 Linux 支持 ext2、vfat、ntfs、yaffs2、ramfs、cramfs 和 romfs 等多种文件系统。在嵌入式系统中常用的闪存设备的文件系统有 cramfs、romfs、ramfs、jffs2、yaffs 等文件系统。

### 5．init目录

init 子目录包含核心的初始化代码（注意，不是系统的引导代码）。它包含两个文件 main.c 和 version.c，这是研究核心如何工作的一个非常好的起点。

### 6．ipc目录

ipc 子目录包含核心进程间的通信代码。Linux 下进程间通信机制主要包括管道、信号、消息队列、共享内存、信号量、套接口。

### 7．kernel目录

kernel 子目录包含内核管理的核心代码。与处理器结构相关代码都放在 arch/*/kernel 目录下。

### 8．net目录

net 子目录里是核心的网络部分代码，其每个子目录存放一个具体的网络协议或者网络模型代码。

### 9．mm目录

mm 子目录包含了所有的内存管理代码。与具体硬件体系结构相关的内存管理代码位于 arch/*/mm 目录下。

### 10．scripts目录

scripts 子目录包含用于配置核心的脚本文件。

### 11．lib目录

lib 子目录包含了核心的库代码，与处理器结构相关的库代码被放在 arch/*/lib/目录下。

## 4.2　内核配置选项

内核配置通常是对内核支持的各个功能进行取舍配置，将配置的方案保存到 configure 文件中。在编译内核的时候，就会根据此配置对内核进行取舍编译。在源码目录下通过 make menuconfig 命令进入内核的配置界面，如图 4.3 所示。在对内核功能进行配置时，使用键盘的方向键移动光标位置，使用 Enter 键选择菜单，使用空格键修改配置选项。

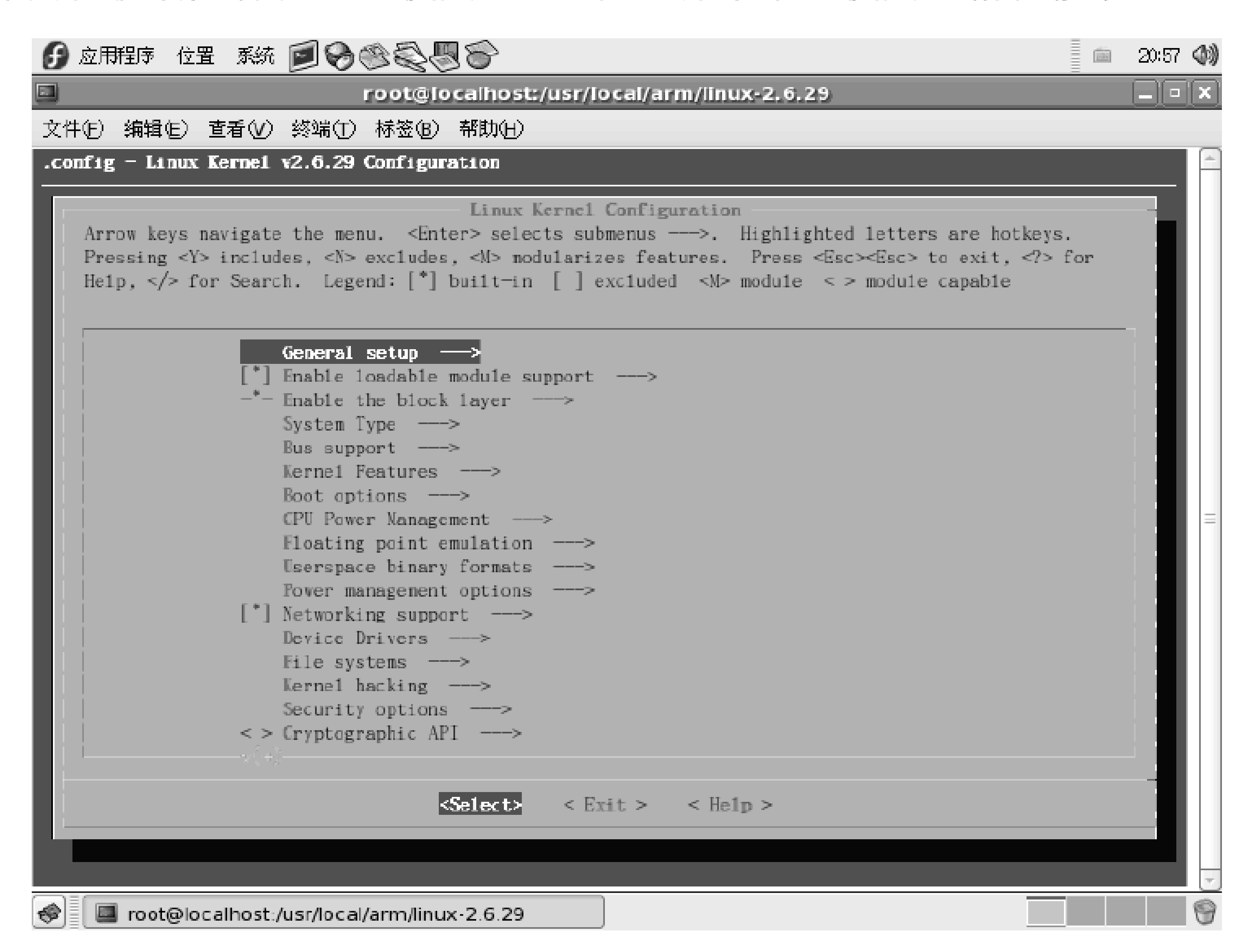



图 4.3　内核配置界面

Linux 配置选项的基本分类和涵义如下。

### 4.2.1　一般选项

菜单选项（General setup）的子菜单中包含一些内核通用配置选项，如表 4.1 所示。在

一般配置选项中如果对系统没有特殊要求，可以只选择 System V IPC 配置。

表 4.1　一般选项

| 选　项　名 | 说　　明 |
|---|---|
| Automatically append version information to the version string | 自动在版本后添加版本信息，编译时需要有 perl 及 git 仓库支持，通常可以不选 |
| Support for paging of anonymous memory (swap) | 支持交换内存，通常选择 |
| System V IPC | 进程间通信，通常需要配置 |
| POSIX Message Queues | POSIX 消息队列，通常需要配置 |
| BSD Process Accounting | 可以将行程资料记录下来，通常建议配置 |
| Export task/process statistics through netlink | 通过 netlink 接口向用户空间导出任务/进程的统计信息 |
| Auditing support | 审计支持，某些内核模块（例如 SELinux）需要配置 |
| RCU subsystem | 同步机制 |
| Kernel .config support | 提供.config 配置文件支持 |
| Kernel log buffer size (16=>64KB, 17=>128KB) | 内核日志缓冲区大小（16 代表 64KB，17 代表 128KB） |
| Group CPU scheduler | CPU 组调度 |
| Control Group support | 控制组支持 |
| Create deprecated sysfs layout for older userspace tools | 为旧的用户空间工具创建过时的文件系统风格 |
| Kernel->user space relay support (formerly relayfs) | 在某些文件系统上（比如 debugfs）提供从内核空间向用户空间传递大量数据的接口 |
| Namespace support | 命名空间支持 |
| Initial RAM filesystem and RAM disk (initramfs/initrd) support | 初始化 RAM 文件系统的源文件。initramfs 可以将根文件系统直接编译进内核，一般是 cipo 文件。对嵌入式系统有用 |
| Optimize for size | 代码优化。如果不了解编译器，建议不选 |
| Configure standard kernel features (for small systems) | 为特殊环境准备的内核选项，通常不需要这些非标准内核 |
| Disable heap randomization | 禁用随机 heap（heap 堆是一个应用层的概念，即堆对 CPU 是不可见的，它的实现方式有多种，可以由 OS 实现，也可以由运行库实现，也可以在一个栈中来实现一个堆） |
| Choose SLAB allocator | 选择内存分配管理器，建议选择 |
| Profiling support | 支持系统评测，建议不选 |
| Kprobes | 探测工具，开发人员可以选择，建议不选 |

## 4.2.2　内核模块加载方式支持选项

菜单选项（Loadable module support）的子菜单中包含一些内核模块加载方式选项，如表 4.2 所示。如果对模块的加载方式有特殊要求，如可以强制卸载正在使用的模块的要求，那么可以配置相关的模块加载方式。

表 4.2　内核模块加载方式

| 选　项　名 | 说　　明 |
|---|---|
| Forced module loading | 允许强制加载模块驱动 |
| Module unloading | 允许卸载已经加载的模块，建议选择 |
| Forced module unloading | 允许强制卸载正在运行的模块，该功能危险，建议不选 |
| Module versioning support | 允许使用其他内核版本的模块，建议不选 |
| Source checksum for all modules | 为所有的模块校验源码，可以不选 |

### 4.2.3　系统调用、类型、特性、启动相关选项

菜单选项（Block layer）的子菜单中包含一些系统调用方式选项，如表 4.3 所示。在配置内核时可以不选该菜单选项。

表 4.3　系统调用方式

| 选　项　名 | 说　　明 |
|---|---|
| Support for Large Block Devices and files | 使用大容量块设备时选择 |
| Support for tracing block io actions | 支持块队列 I/O 跟踪 |
| Block layer SG support v4 | 支持通用 scsi 块设备第 4 版 |
| Block layer data integrity support | 支持块设备数据完整性 |
| IO Schedulers | I/O 调度器 |

菜单选项（System Type）的子菜单中包含一些系统类型选项，在配置内核时直接选择对应的芯片类型即可。对特定的平台选择相应的支持类型。

菜单选项（Kernel Features）的子菜单中包含一些系统特性选项，如表 4.4 所示。在嵌入式系统中，一般不对这些选项进行配置。

表 4.4　系统特性

| 选　项　名 | 说　　明 |
|---|---|
| Preemptible Kernel | 抢占式内核。建议采用 |
| Use the ARM EABI to compile the kernel | 使用 ARM EABI 编写内核 |
| Allow old ABI binaries to run with this kernel | 使内核支持旧版本的 ABI 程序 |
| Memory model | 只有 Flat Memory 供选择 |
| Add LRU list to track non-evictable pages | 对没有使用的页采用最近最少使用算法，建议选择 |

菜单选项（Boot Options）的子菜单中包含一些系统启动选项，如表 4.5 所示。

表 4.5　系统启动

| 选　项　名 | 说　　明 |
|---|---|
| (0)Compressed ROM boot loader base address | xImage 存放的基地址 |
| (0)Compressed ROM boot loader BSS address | BSS 地址 |
| ()Default Kernel command string | 内核启动参数 |
| Kernel Execute-In-Place from ROM | 从 ROM 中直接运行内核，该内核使用 make xipImage 编译 |
| (0x00080000)XIP Kernel Physical Location | 选择 XIP 后，内核存放的物理地址 |
| Kexec system call | Kexec 系统呼叫 |

### 4.2.4　网络协议支持相关选项

菜单选项（Networking Support）的子菜单中包含一些网络协议支持的选项，如表 4.6 所示。基本只需要在 Networking options 子菜单中选择具体所需的网络协议即可。

**表 4.6　网络协议**

| 选　项　名 | 说　　明 |
|---|---|
| Networking options | 该菜单的子菜单包含支持的各种具体网络协议，在开发中可以根据需要进行配置 |
| Amateur Radio support | 业余无线电支持，一般不选 |
| CAN bus subsystem support | CAN 总线子系统支持 |
| IrDA (infrared) subsystem support | 红外线支持 |
| Bluetooth subsystem support | 蓝牙支持 |
| RxRPC session sockets | RxRPC 会话套接字支持 |
| Phonet protocols family | Phonet 协议族支持 |
| Wireless | 无线电协议支持 |
| WiMAX Wireless Broadband support | WiMAX 无线宽带支持 |
| RF switch subsystem support | RF 交换子系统支持 |
| Plan 9 Resource Sharing Support (9P2000) | 9 计划资源共享支持 |

### 4.2.5　设备驱动支持相关选项

菜单选项（Device drivers）的子菜单中包含一些设备驱动的选项，如表 4.7 所示。重点说明了 MTD 设备相关的驱动。需要支持设备驱动时可以配置相关的选项。

**表 4.7　设备驱动**

| 选　项　名 | 说　　明 |
|---|---|
| Connector - unified userspace <-> kernelspace linker | 用户空间和内核空间的统一连接器 |
| Memory Technology Devices (MTD) support | MTD 设备支持，嵌入式系统使用 |
| ->Debugging | 调试功能 |
| ->MTD concatenating support | 连接多个 MTD 设备，例如使用 JFFS2 文件系统管理多片 Flash 的情形。只有一片 Flash 时不选 |
| ->MTD partitioning support | Flash 分区支持，建议选择 |
| ->MTD tests support | MTD 测试支持 |
| ->RedBoot partition table parsing | 使用 RedBoot 解析 Flash 分区表，如果需要读取这个分区表的信息，选择此项 |
| ->Command line partition table parsing | 允许通过内核命令行传递 MTD 分区表信息 |
| ->ARM Firmware Suite partition parsing | 使用 AFS 分区信息 |
| ->TI AR7 partitioning support | AR7 分区支持 |
| ->Direct char device access to MTD devices | 将系统中的 MTD 设备看作字符设备进行读/写 |

续表

| 选　项　名 | 说　　明 |
|---|---|
| ->Caching block device access to MTD devices | 文件系统挂载后，模拟块设备进行访问。常用于只读文件系统。如果是 DiskOnChip 使用 NFTL 方式 |
| ->FTL (Flash Translation Layer) support | 提供对 Flash 翻译层支持，可以不选 |
| ->NFTL (NAND Flash Translation Layer) support | NAND Flash 翻译层支持，可以不选 |
| -> INFTL (Inverse NAND Flash Translation Layer) support | 提供 INFTL 支持，DiskOnChip 使用 |
| -> Resident Flash Disk (Flash Translation Layer) support | 提供 RFD 支持，为嵌入式系统提供类似 BIOS 功能 |
| -> NAND SSFDC (SmartMedia) read only translation layer | NAND SSFDC 只读翻译层 |
| -> Log panic/oops to an MTD buffer | MTD 缓冲区日志 |
| -> RAM/ROM/Flash chip drivers | RAM/ROM/Flash 芯片驱动 |
| ->Mapping drivers for chip access | 为芯片的访问方式选择 Mapping 驱动 |
| -> Self-contained MTD device drivers | 自身包含 MTD 设备驱动，一般不选 |
| ->NAND Device Support | NAND Flash 支持 |
| ->OneNAND Device Support | One NAND 相关驱动 |
| ->LPDDR flash memory drivers | LPDDR Flash 内存驱动 |
| ->UBI - Unsorted block images | 只提供 UBI 支持 |
| Parallel port support | 并口支持 |
| Block devices | 红外线支持 |
| Bluetooth subsystem support | 蓝牙支持 |
| RxRPC session sockets | RxRPC 会话套接字支持 |
| Phonet protocols family | Phonet 协议族支持 |
| Wireless | 无线电协议支持 |
| WiMAX Wireless Broadband support | WiMAX 无线宽带支持 |
| RF switch subsystem support | RF 交换子系统支持 |
| Plan 9 Resource Sharing Support (9P2000) | 9 计划资源共享支持 |

## 4.2.6　文件系统类型支持相关选项

菜单选项（File Systems）的子菜单中包含一些文件系统配置的选项，如表 4.8 所示。内核移植完成后，通常需要制作文件系统，可以在此部分选择内核支持的文件系统格式。

**表 4.8　文件系统**

| 选　项　名 | 说　　明 |
|---|---|
| Second extended fs support | Ext2 文件系统支持 |
| Ext3 journalling file system support | Ext3 文件系统支持 |
| The Extended 4 (ext4) filesystem | Ext4 文件系统支持 |
| Reiserfs support | Reiserfs 文件系统支持 |

续表

| 选　项　名 | 说　　明 |
|---|---|
| JFS filesystem support | JFS 文件系统支持 |
| XFS filesystem support | XFS 文件系统支持 |
| OCFS2 file system support | OCFS2 文件系统支持 |
| Btrfs filesystem (EXPERIMENTAL) Unstable disk format | Btrfs 文件系统，不稳定，建议不选择 |
| Dnotify support | 文件系统变化通知机制支持 |
| Inotify file change notification support | Inotify 是 Dnotify 的替代者，在高版内核中默认支持 |
| Quota support | 磁盘限额支持 |
| Kernel automounter support | 自动挂载远程文件系统，如 NFS |
| Kernel automounter version 4 support (also supports v3) | 自动挂载远程文件系统，对版本 4 和版本 3 都支持 |
| FUSE (Filesystem in Userspace) support | 在用户空间挂载文件系统，建议选择 |
| CD-ROM/DVD Filesystems | ISO 9660，UDF 等文件系统支持 |
| DOS/FAT/NT Filesystems | FAT/NTFS 文件系统支持。如果用于访问存储设备，并且包含像 Windows 文件时选上该选项 |
| Pseudo filesystems | 伪操作系统，多指内存中的操作系统 |
| Miscellaneous filesystems | 杂项文件系统，包括 ADFS，BFS，BeFS，HPFS 等，比较少用，建议不选 |
| Network File Systems | 网络文件系统。其中只有 NFS 在产品开发过程中用。在开发过程可以选用 |
| Partition Types | 分区类型。该菜单下提供很多中类型，但在嵌入式产品中很少用，建议不选 |
| Distributed Lock Manager (DLM) | 分布式锁管理器 |

## 4.2.7　安全相关选项

菜单选项（Security options）的子菜单中包含一些安全配置选项。很少用，建议不选。菜单选项（Kernel hacking）的子菜单中包含内核黑客配置选项。建议不选。菜单选项（Cryptographic API）的子菜单中包含内核加密算法配置选项。很少用，建议不选。

## 4.2.8　其他选项

菜单选项（Bus Support）的子菜单中包含一些总线接口支持，嵌入式系统可以不选。菜单选项（CUP Power Management）的子菜单中包含电源管理选项，嵌入式系统可以不选。菜单选项（Floating）的子菜单中包含一些总线接口支持，嵌入式系统可以不选。菜单选项（Library routines）的子菜单中包含一些库配置选项，主要提供 CRC 支持，在开发通信类产品时可以选择对应的 CRC。

# 4.3　内核裁剪及编译

经过对内核的认识和对裁剪配置项的了解，接下来实际操作。针对 S3C2440 开发板进行裁剪 Linux 内核。

## 4.3.1　安装内核源代码

在前面章节中已经介绍了建立交叉编译环境。如果还没有建立编译环境，请参考相关章节。获得源码可以直接从网上下载开发板对应的源码。该源码相比 Linux 基本内核源码增加了对应平台相关的内容。将源代码压缩包复制到/usr/local/arm 目录下，使用 tar 命令解压源码。

```
tar -zxvf linux-2.6.29-HY2440.tgz
```

tar 命令带上 zxvf 参数可以看到详细的解压过程，如图 4.4 所示。

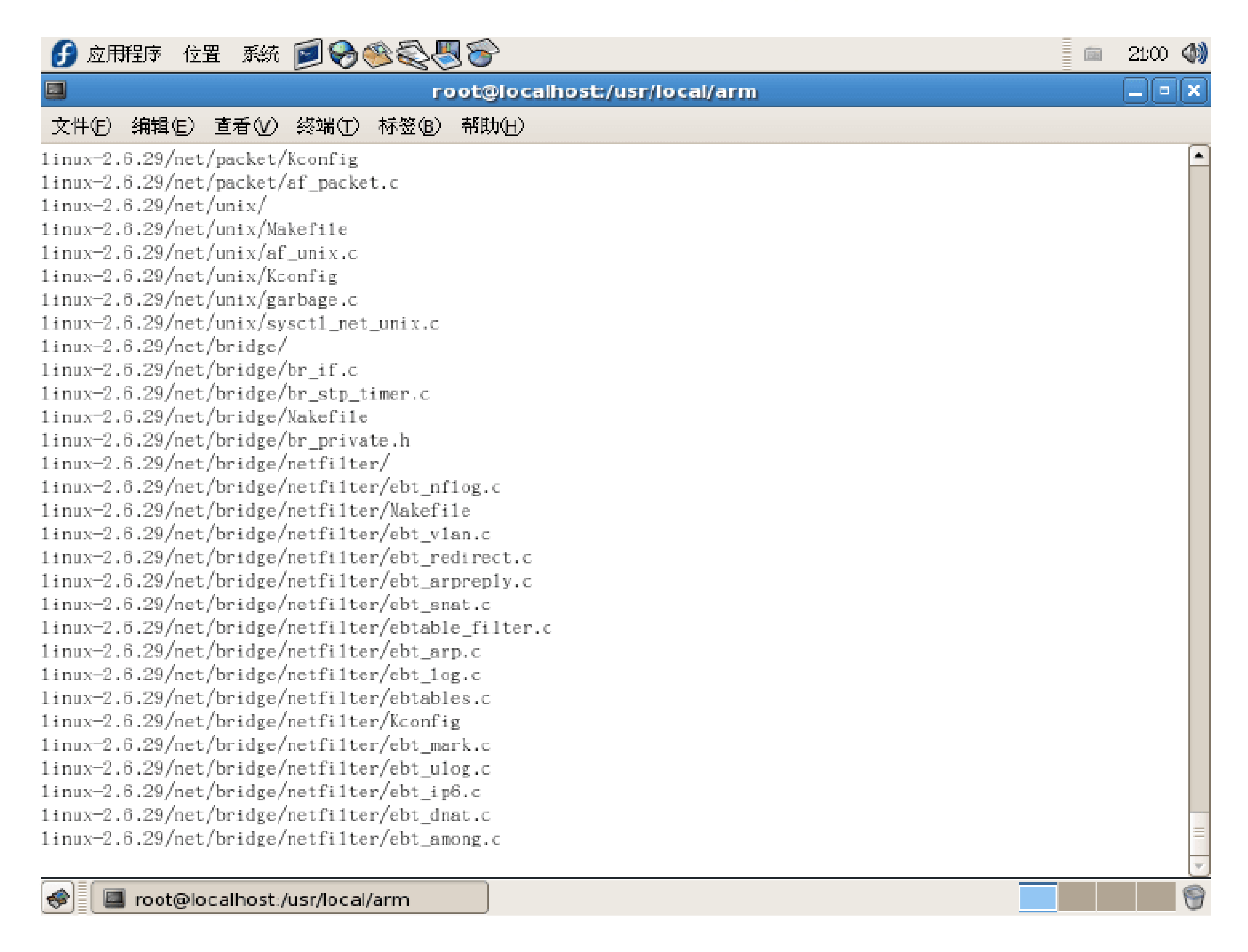



图 4.4　内核解压过程

## 4.3.2　检查编译环境设置

源代码解压完成后，进入 linux-2.6.29 目录下，然后使用 VI 命令编辑 Mackfile。确定编译环境为 arm 交叉编译工具与本机安装的路径和一致。

```
ARCH = arm
CROSS_COMPILE = /usr/local/arm/4.3.2/bin/arm-linux-
```

## 4.3.3　配置内核

使用 make menuconfig 命令进入内核配置界面，如图 4.3 所示。注意在 linux-2.6.29 目录下，执行 make menuconfig 命令才能正确进入配置界面。下面给出一个内核的基本配置。

（1）在一般 General setup 配置项中选择子项 System V IPC。由于要支持处理器在程序之间同步和交换信息，如果不选这项，很多程序将运行不起来，所以选择 General setup 配置项中的子项 System V IPC，其他可以不选，如图 4.5 所示。在此配置界面中还有一个选项[*] Initial RAM filesystem and RAM disk (initramfs/initrd) support 在制作 Ramdisk 文件系统时，应该选上该选项，如图 4.6 所示。

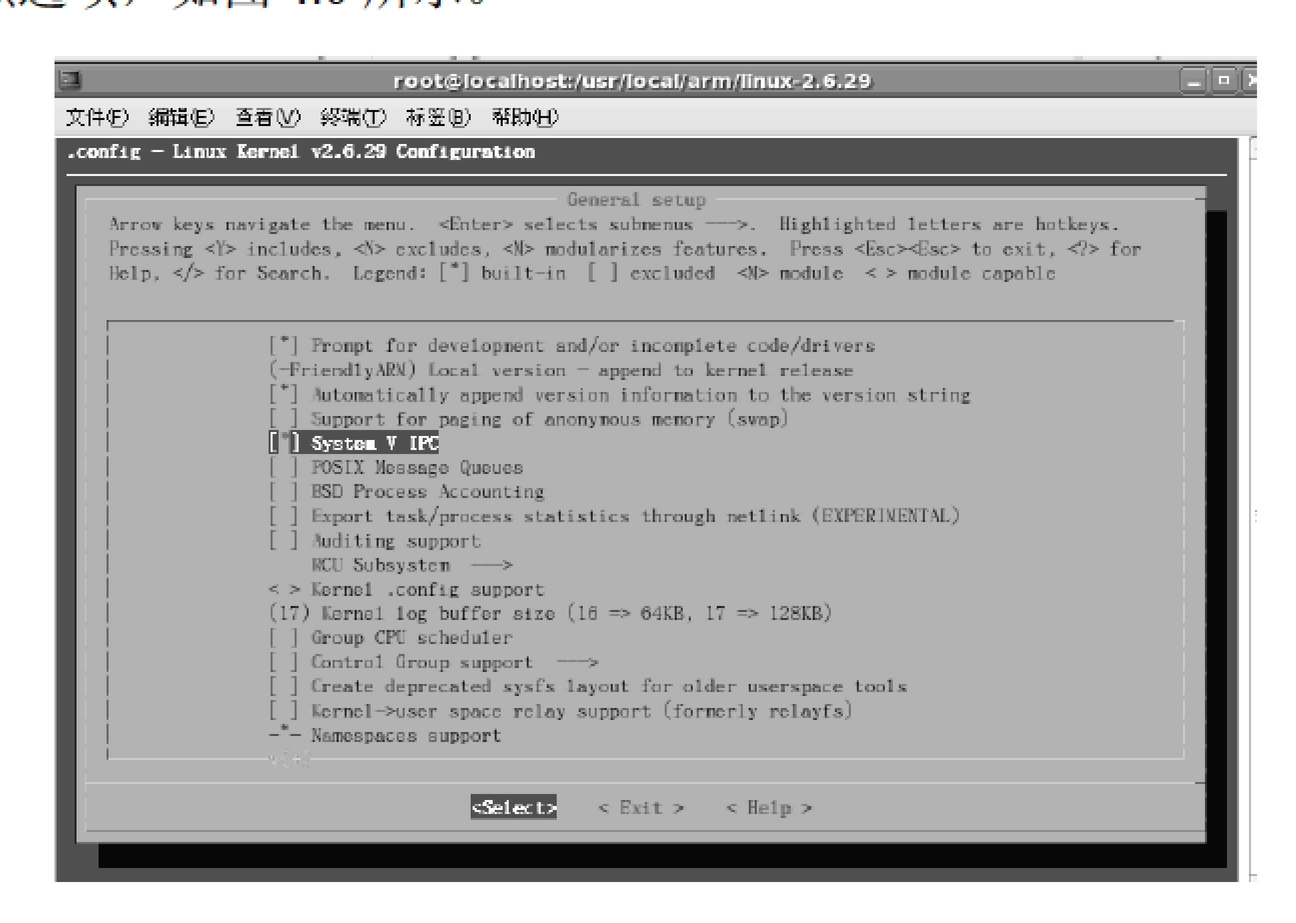



图 4.5　配置 System V IPC

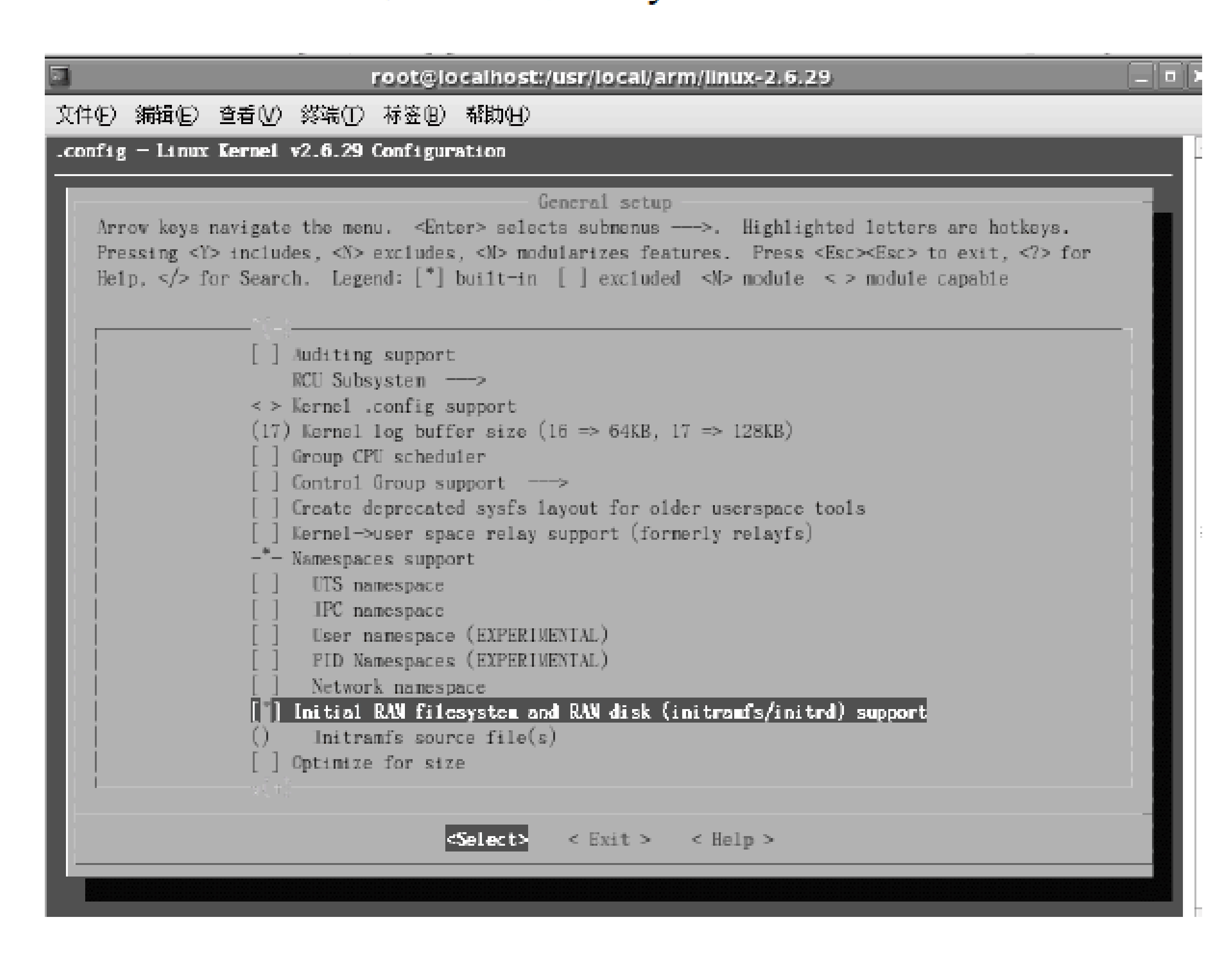



图 4.6　配置 RAM disk 支持

（2）在模块加载方式中，只选择子项 Module unloading，其他可以不选。因为 Force module loading 和 Force module unloading 会造成安全隐患，所以一般不选。Module unloading 支持动态卸载模块，减少内核占用的资源。如图 4.7 所示模块加载方式选项配置。

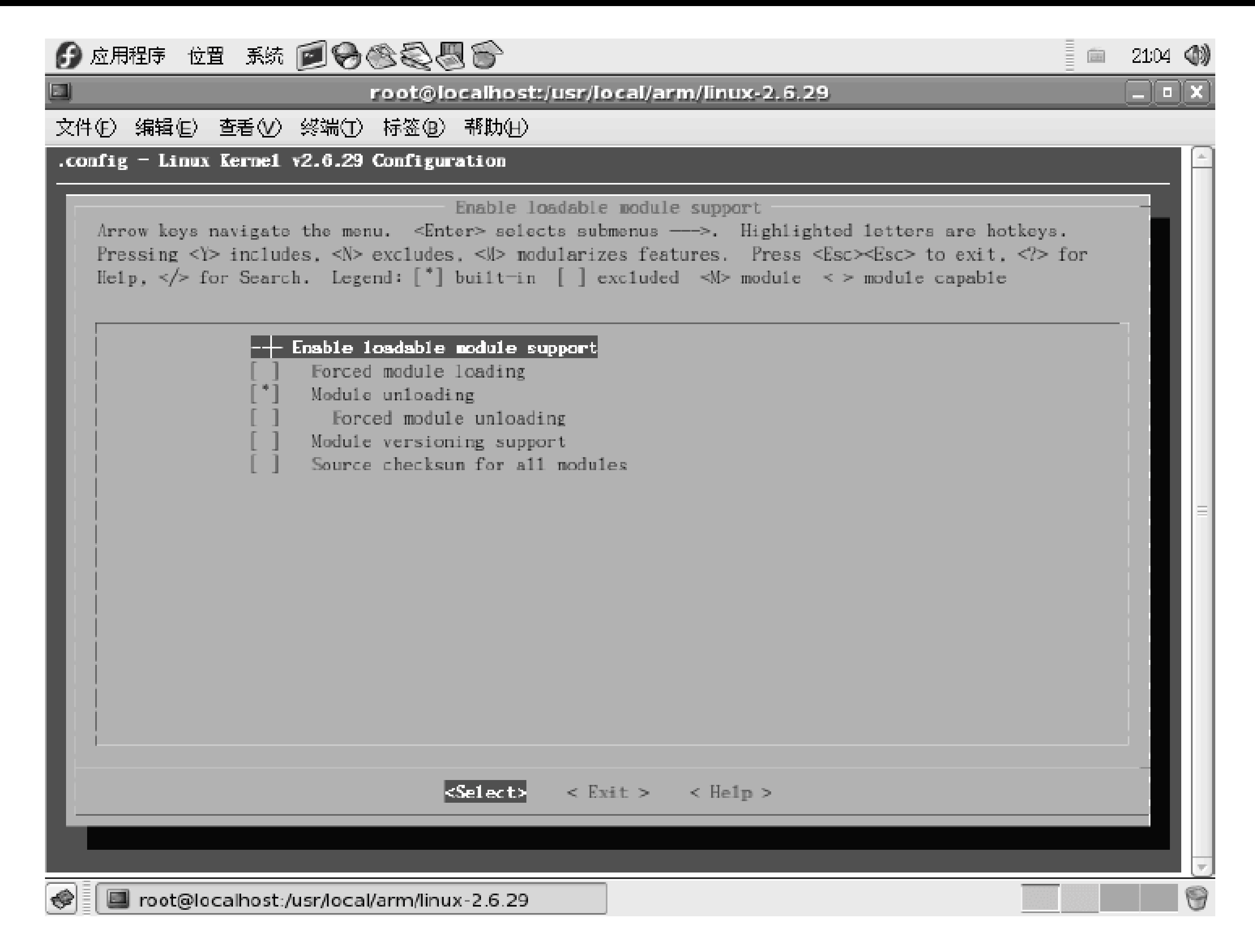



图 4.7　模块加载方式选项配置

（3）如果系统没有对磁盘调度方式有特殊的要求，对 block layer 可以不作任何配置。

（4）在系统类型中选择 S3C3410 DMA support 和 Force UART FIFO on during boot process，选 DMA support 选项是为了支持 2440 直接内存访问。选 UART FIFO 可以支持一般的串口通信协议。如图 4.8 所示为系统类型选项配置。

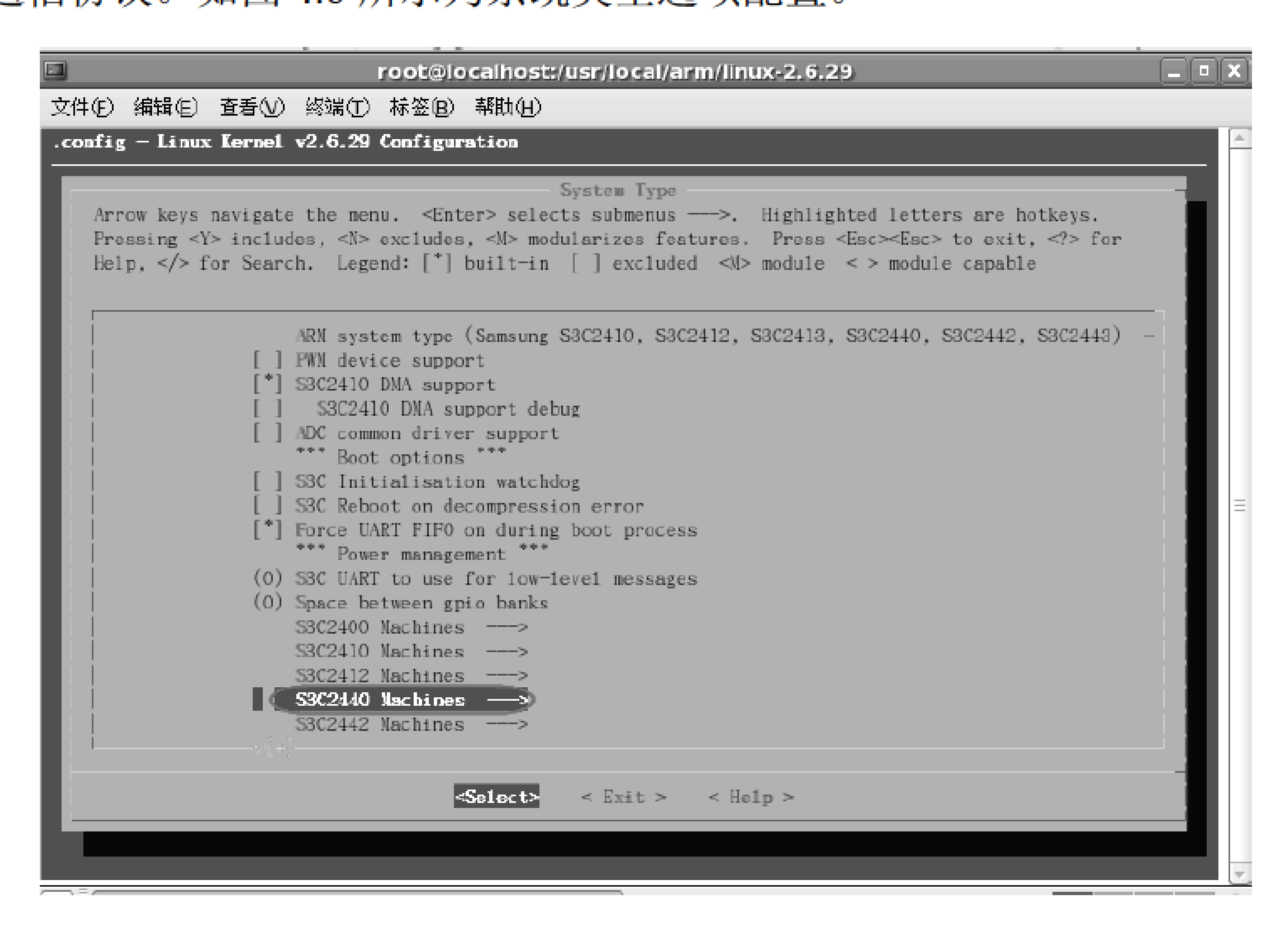



图 4.8　系统类型选项配置

选择 S3C2440 Machines 进入 S3C2440 Machines 的配置界面，选择对应开发板类型的支持，笔者的开发板为 Mini2440，则对应的配置如图 4.9 所示。

（5）对于总线支持 Bus support 配置，一般情况下该选项可以不作配置，除非在开发对应的驱动时。




图 4.9　选择对应的开发板类型

（6）在对系统特性选项进行配置时，建议对选项 Use the ARM EABI to compile the kernel 和选项 Allow old ABI binaries to run with this kernel（EXPERIMENTAL）（NEW）进行配置，如图 4.10 所示。如果交叉编译工具的版本为 arm-linux-gcc4.3.2 时，没有对这两个选项进行配置，就会在烧写完文件系统后出现系统无法启动的错误，错误提示为 Kernel panic - not syncing: Attempted to kill init!

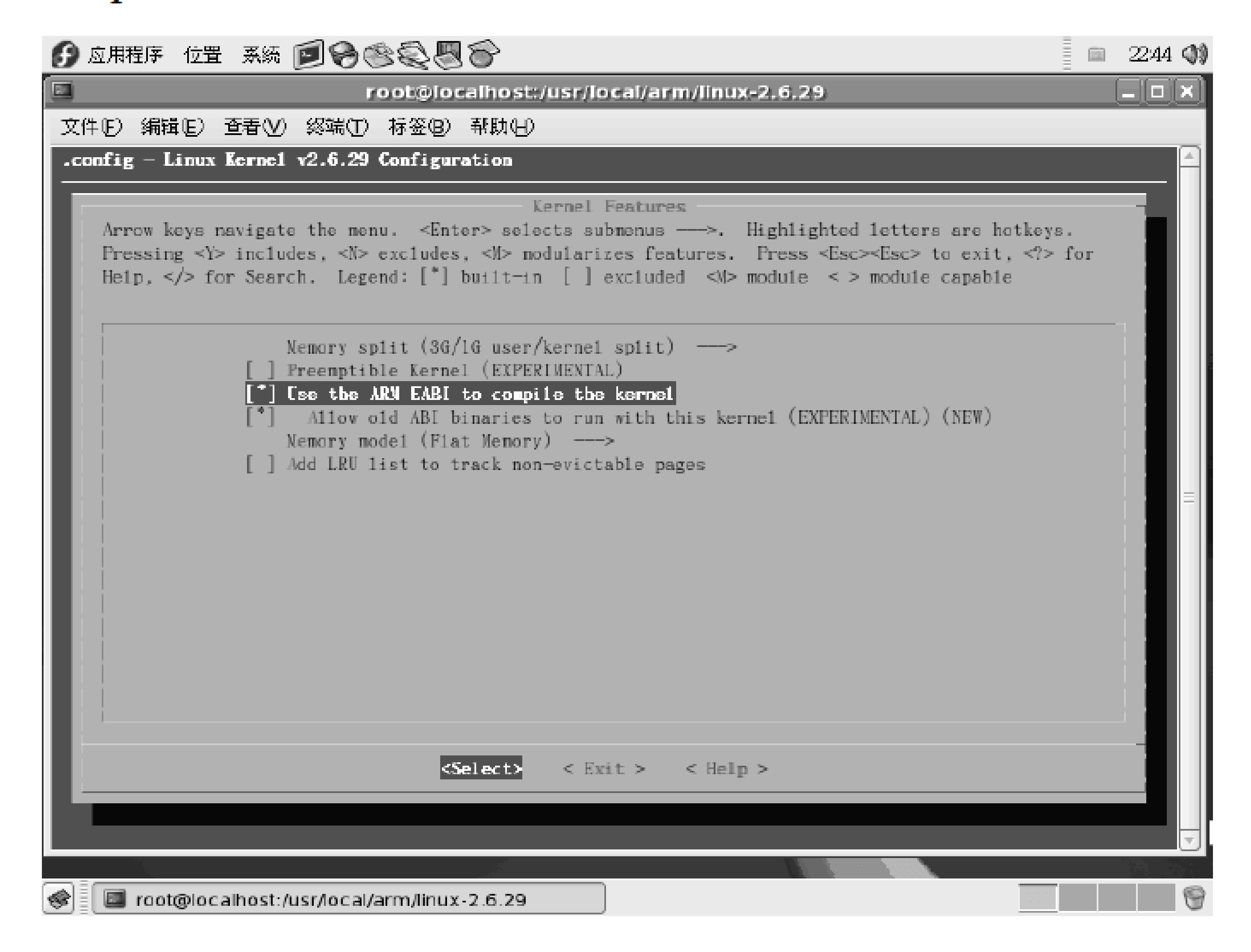



图 4.10　系统类型选项配置

注意：ARM EABI 有许多革新之处，其中最突出的改进就是 Float Point Performance，它使用 Vector Float Point（矢量浮点），因此可以极大提高涉及浮点运算程序的运算速度。如果编译内核的编译器支持 EABI，则在内核中也应该选择对该项的支持。

（7）对启动参数的配置，Bootloader 启动后会将板子的信息、Ramdisk 大小、命令行

字符串等信息传递给内核，然后开始启动内核，文件系统为 Ramdisk 时一般要配置该选项，对选项的具体地址和参数应该根据具体板子、内核大小、文件系统大小来定，该配置界面如图 4.11 所示。

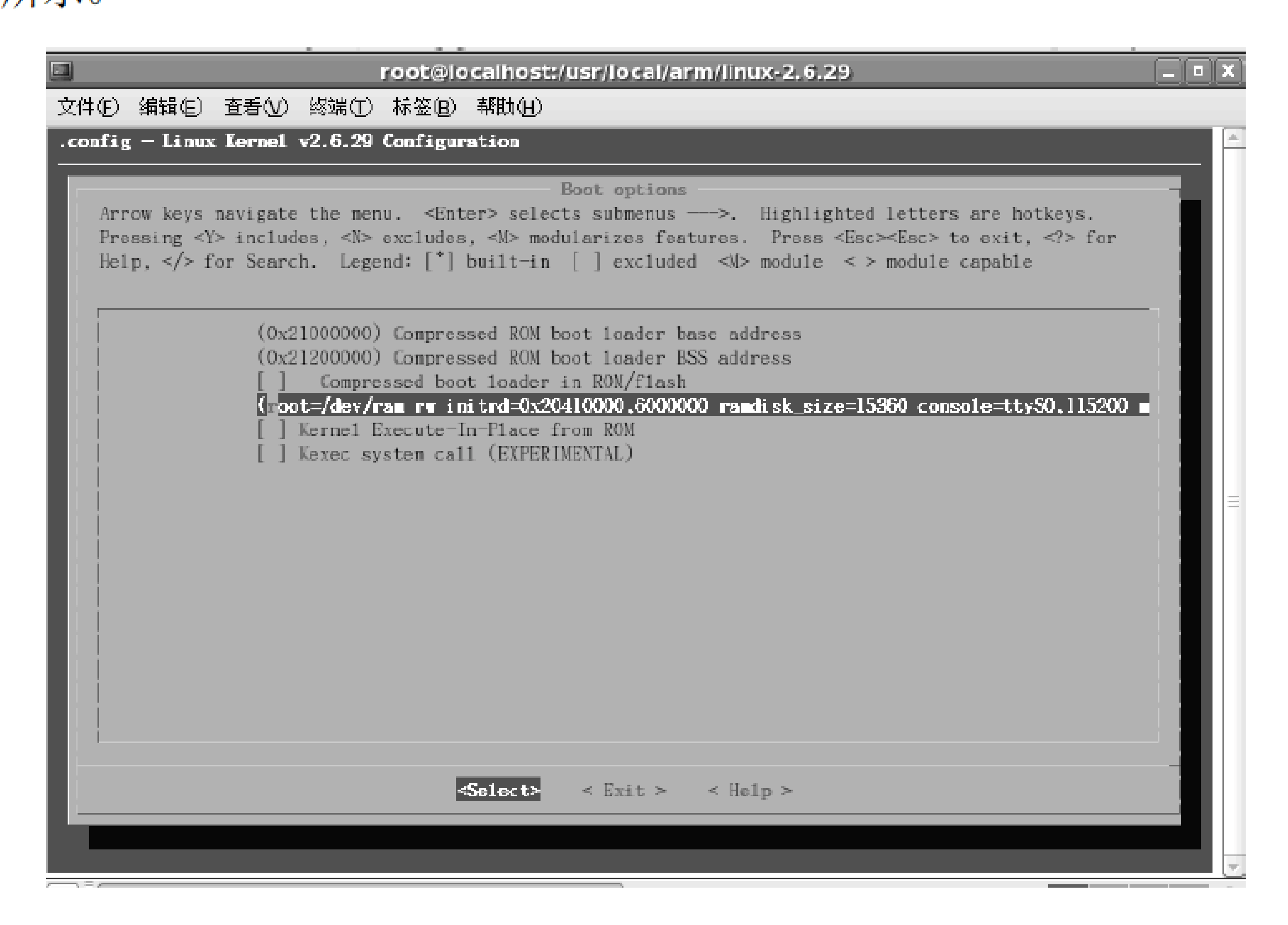



图 4.11 启动参数配置

（8）选项 CPU Power Management 一般不作配置。

（9）选项 Floating point emulation 一般不作配置。

（10）选项 Userspace binary formats，配置 Kernel support for ELF binaries，如图 4.12 所示。

（11）对于电源管理选项一般不作配置。




图 4.12 配置 Kernel support for ELF binaries 选项

（12）对于网络选项的支持，配置 Networking options 中的 TCP/IP networking 和 Unix domain sockets，配置如图 4.13 所示。在 Networking support 下的其他选项，在开发对应的驱动时将对应的选项选上。

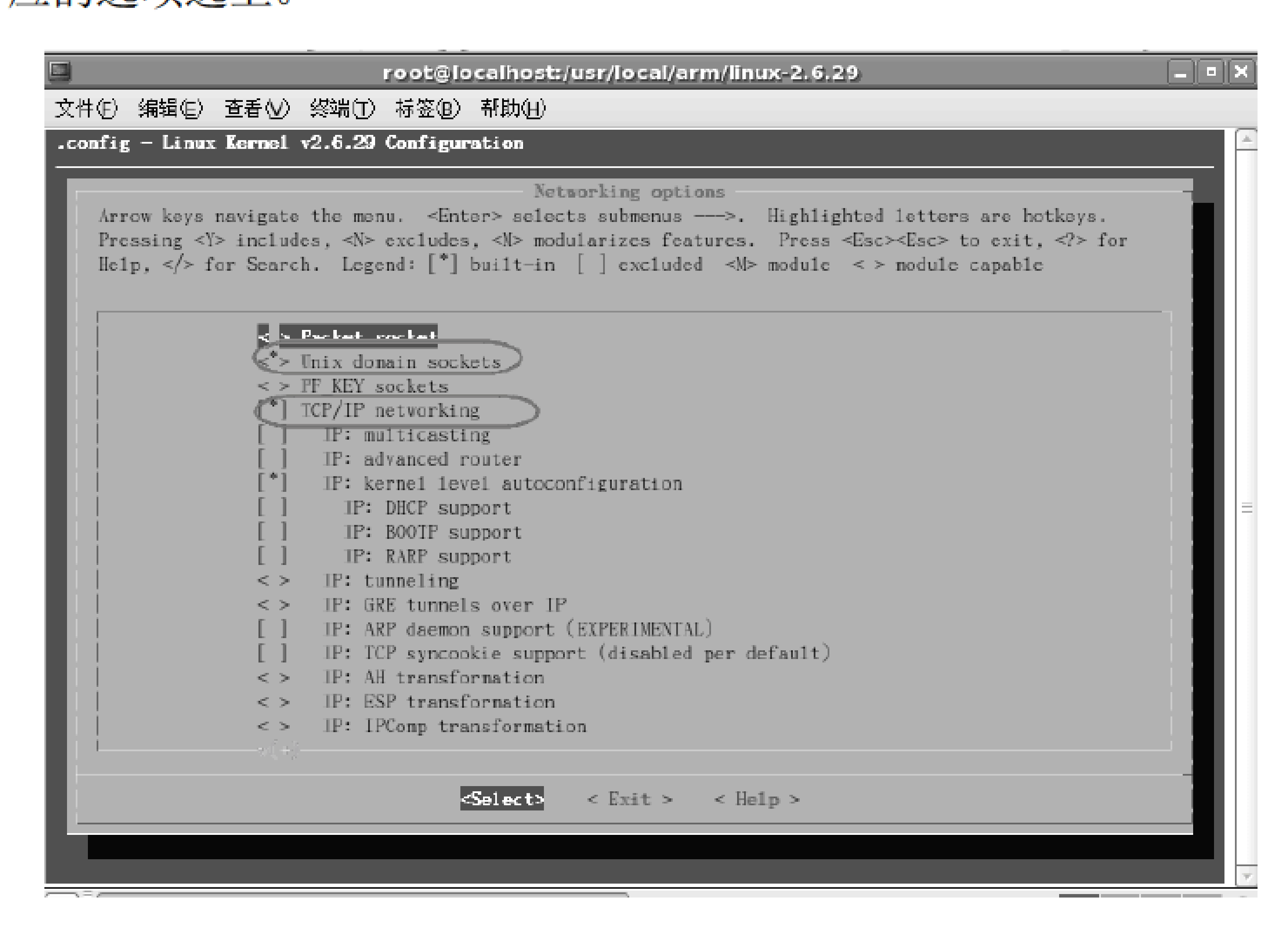



图 4.13　配置 Networking options

（13）设备驱动选择，设备驱动选项是最复杂也是用得最多的配置选项，特别是在开发驱动和系统移植的时候。

在设备驱动选项中添加 MTD 支持，配置 MTD partitioning support 和 Direct char device access to MTDdevices。配置 MTD partitioning support 是支持对 Flash 分区的支持，配置 Direct char device access to MTDdevices 是支持将系统中的 MTD 设备看作字符设备进行读/写，如图 4.14 所示为驱动选项配置。

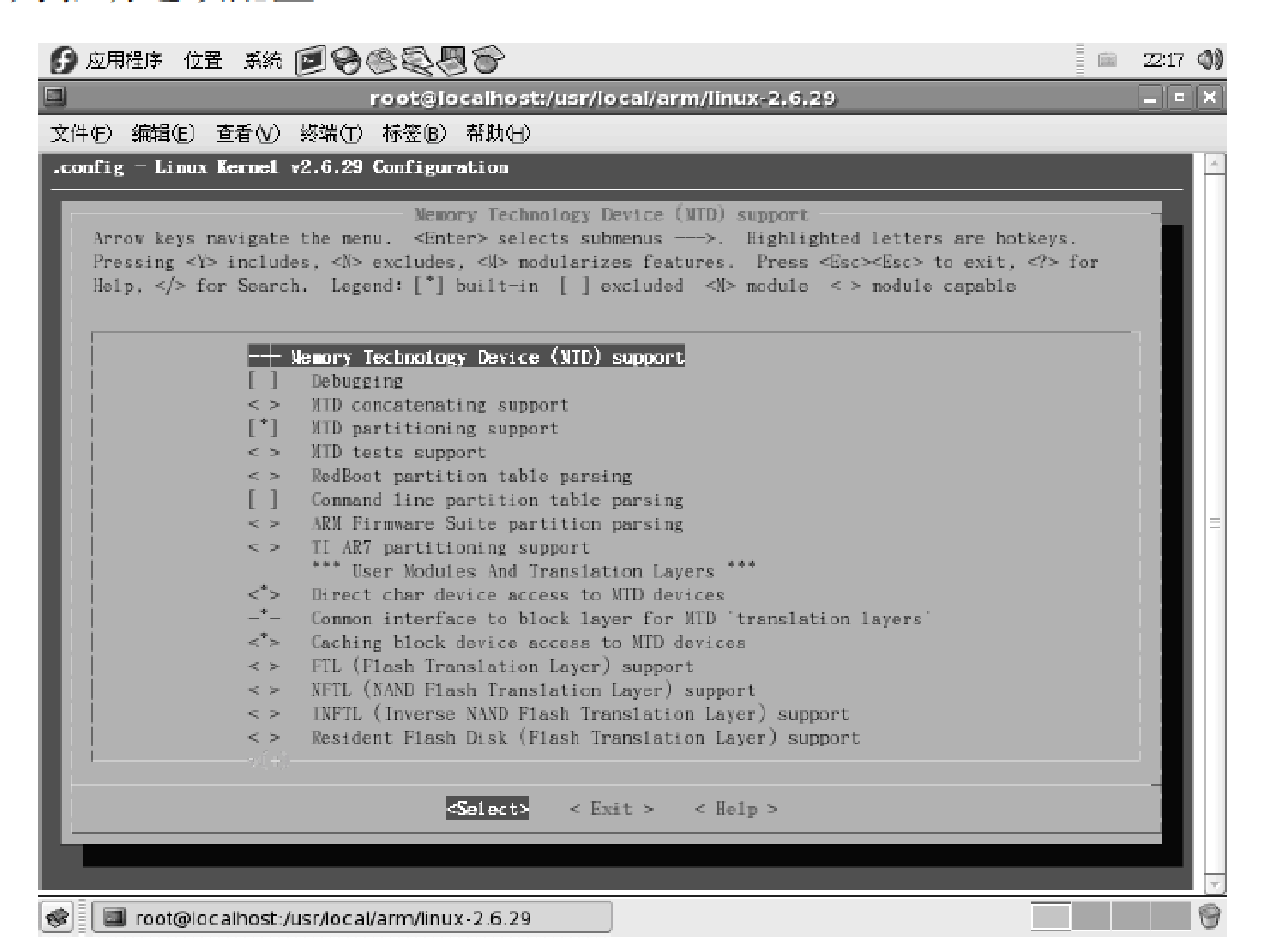



图 4.14　驱动选项配置

在网路设备驱动配置窗口中，为了使开发板支持网卡驱动选择 Ethernet（10 or 100Mbit），如果希望支持 ppp 拨号还可以选择对 ppp 协议的支持等，可以根据具体的开发进行配置，如图 4.15 所示，进入 Ethernet（10 or 100Mbit）配置中选择对应的网卡驱动，如图 4.16 所示。

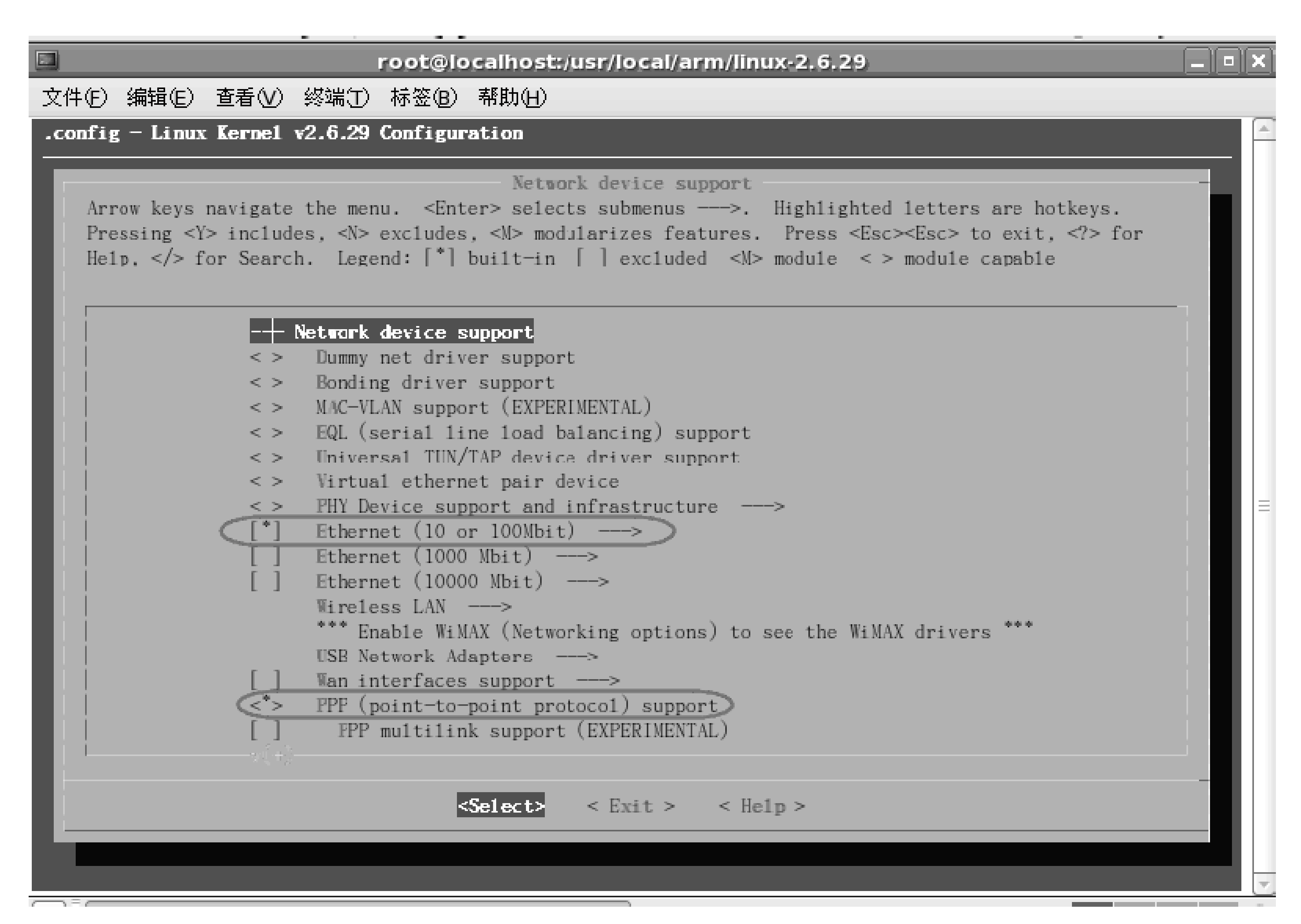



图 4.15　配置网络协议的支持

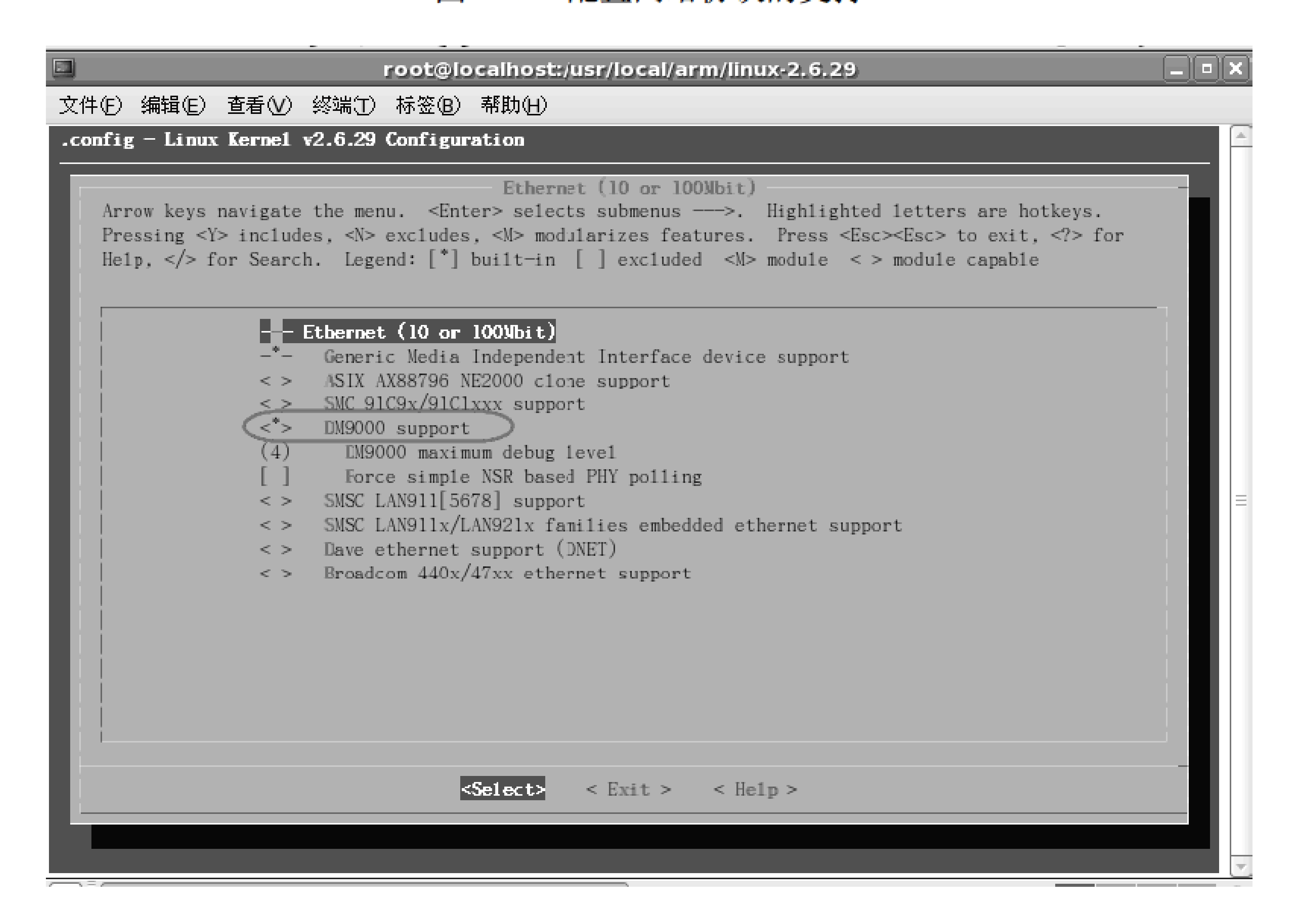



图 4.16　配置对 DM9000 支持

在音频设备驱动时，应该配置 Sound card support，在该配置窗口下有 OSS 驱动框架和 ALSA 驱动框架，其配置界面如图 4.17 所示。

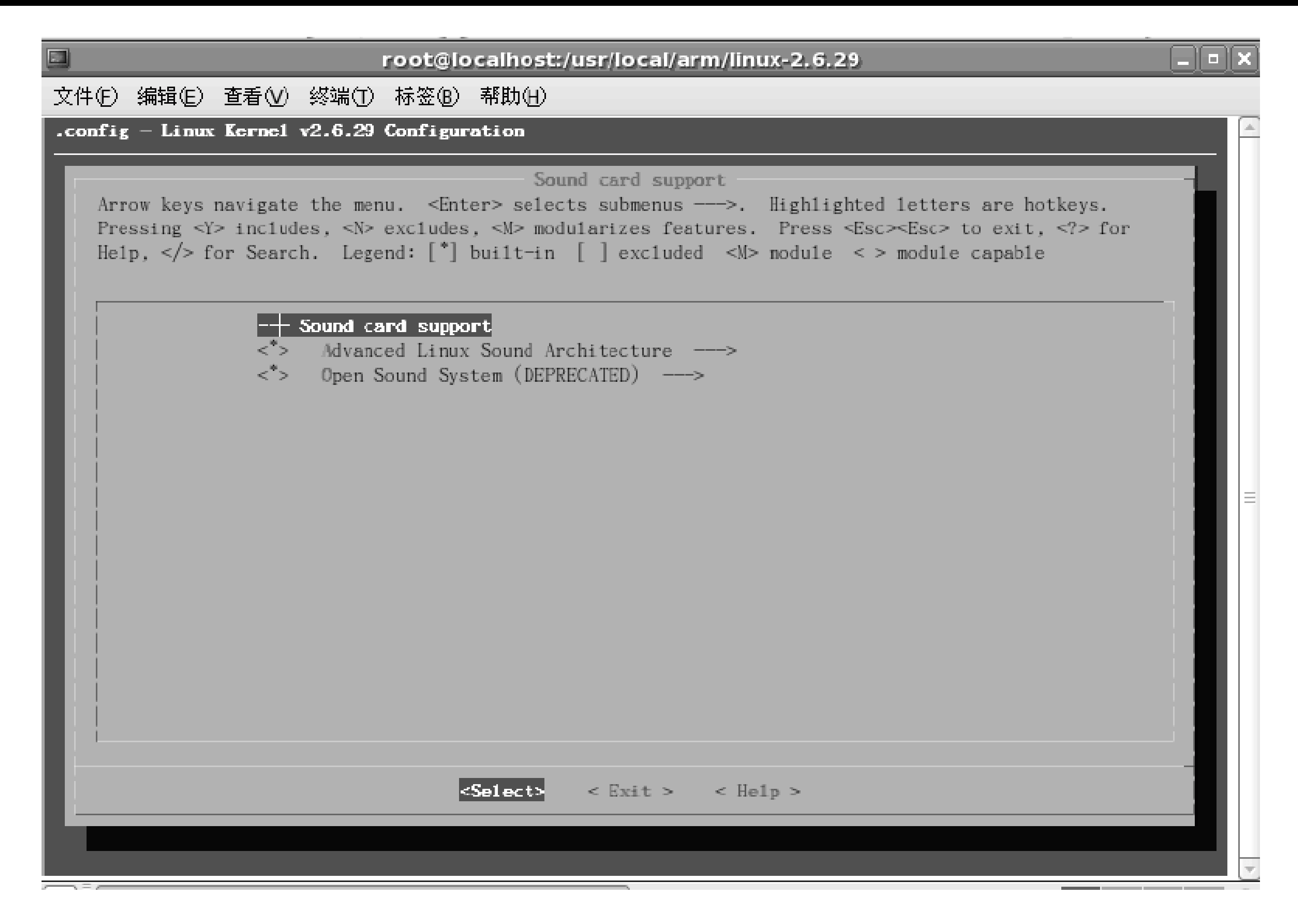



图 4.17　配置声卡驱动的支持

在对驱动框架 Advanced Linux Sound Architecture 进行配置时，如果音频应用程序需要支持数字音频接口、混音接口，则需要配置 OSS Mixer API 和 OSS PCM（digital audio）API，如图 4.18 所示。另外还要对开发板具体的芯片支持，如 Mini2440 采用的 UDA134x，则还要对具体的芯片驱动进行配置，如图 4.19 所示。当然在配置具体音频驱动支持前应该先在内核代码中添加相应的驱动。

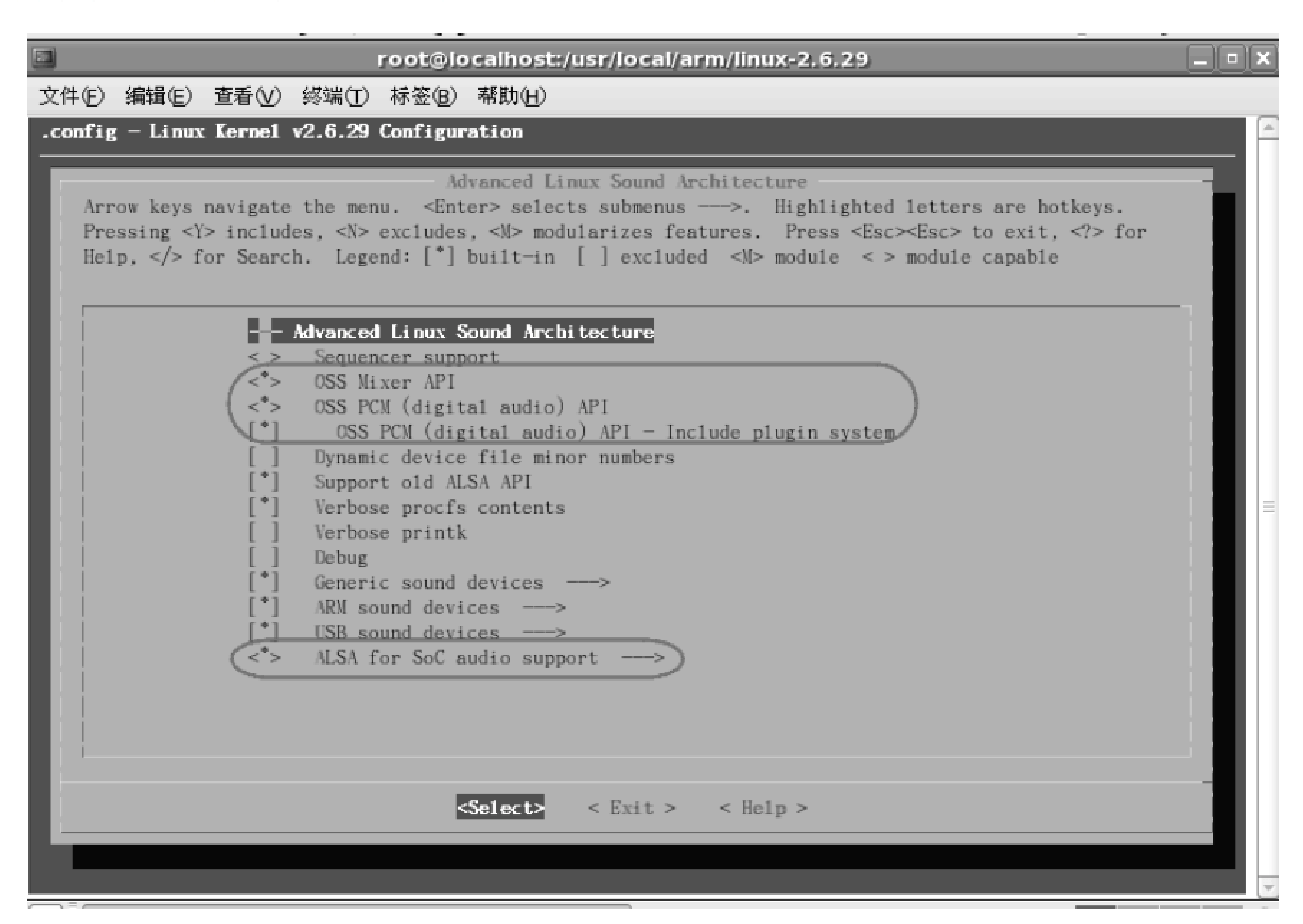



图 4.18　对数字音频接口和混音接口支持

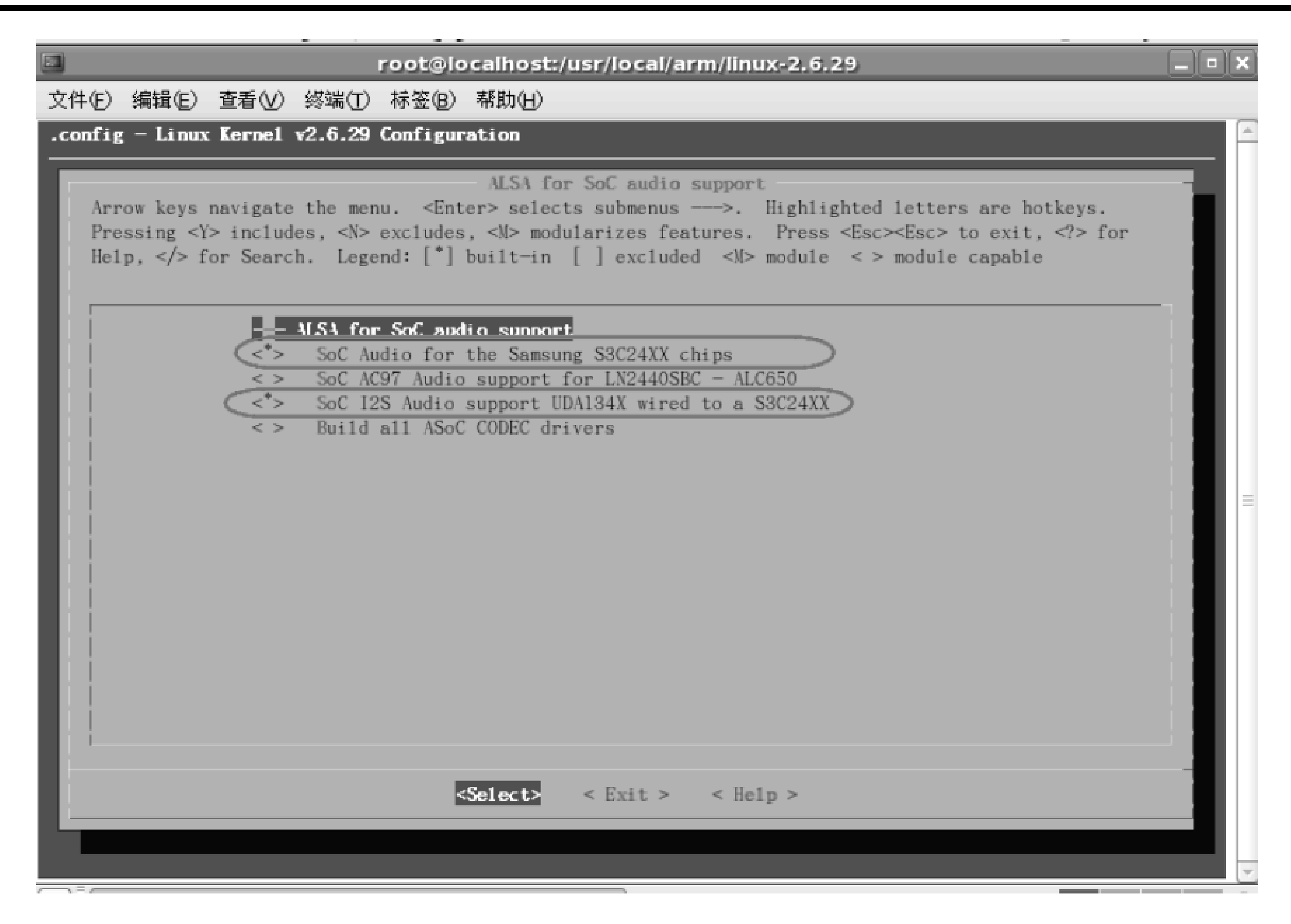



图 4.19　对具体芯片和驱动的支持

USB 设备驱动，也是应该要用到的内核配置选项，在开发 USB 主机驱动时应该配置 OHCI HCD support 选项，在开发 USB 存储设备驱动时配置 USB Mass Storage support 选项，如图 4.20 所示。

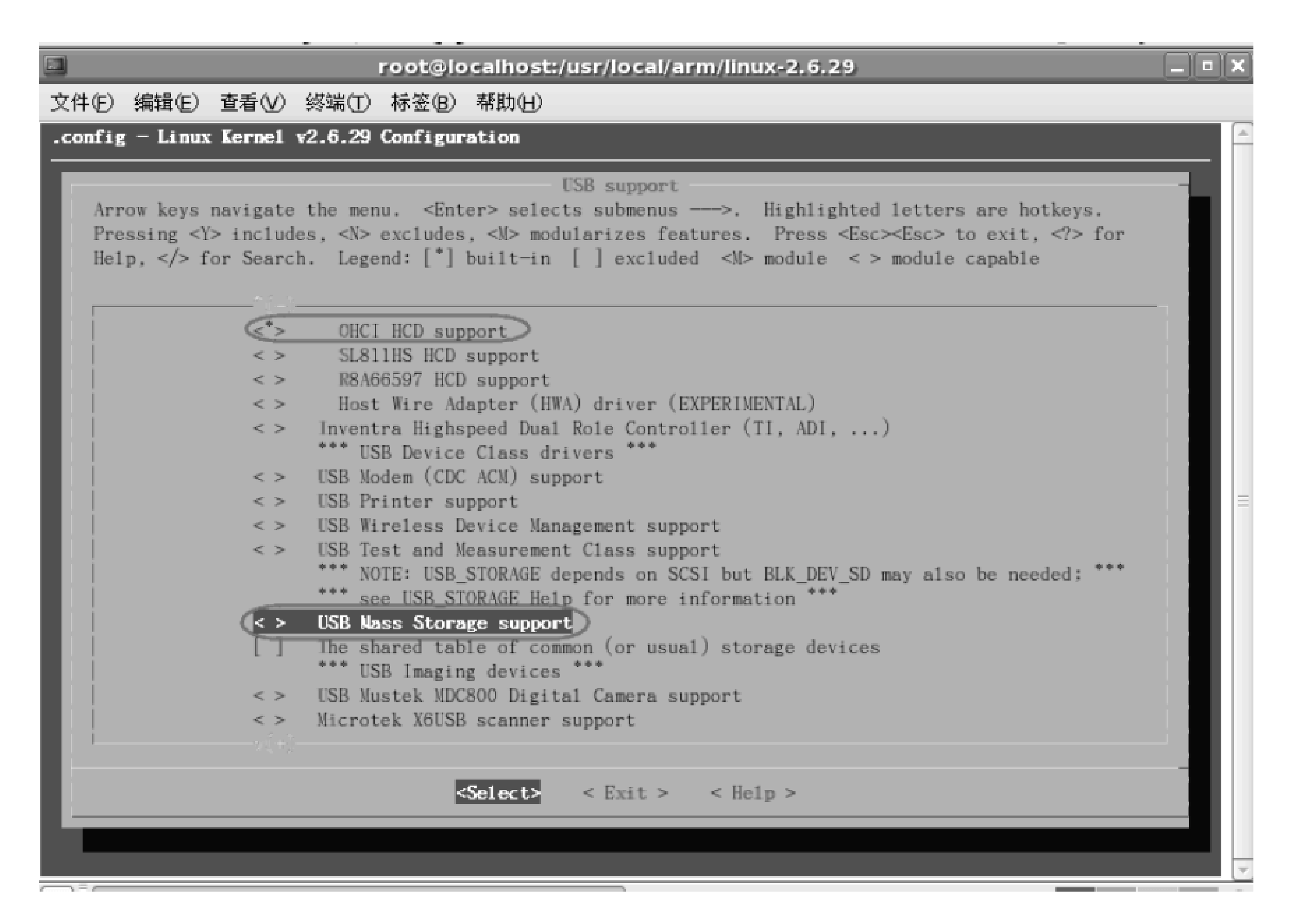



图 4.20　USB 设备驱动配置

在开发键盘、鼠标等输入设备驱动时，应该配置 HID Devices 选项。在开发 SD 卡驱动时应该配置 MMC/SD/SDIO card support 选项。

（14）文件系统选择也是比较重要的部分，在文件系统配置选项时，应该根据所用的文件系统来添加对应的文件系统支持。笔者用到了网络文件系统和 YAFFS2 文件系统，在内核中添加对 NFS 和 YAFFS2 文件系统的支持，如图 4.21 和图 4.22 所示。

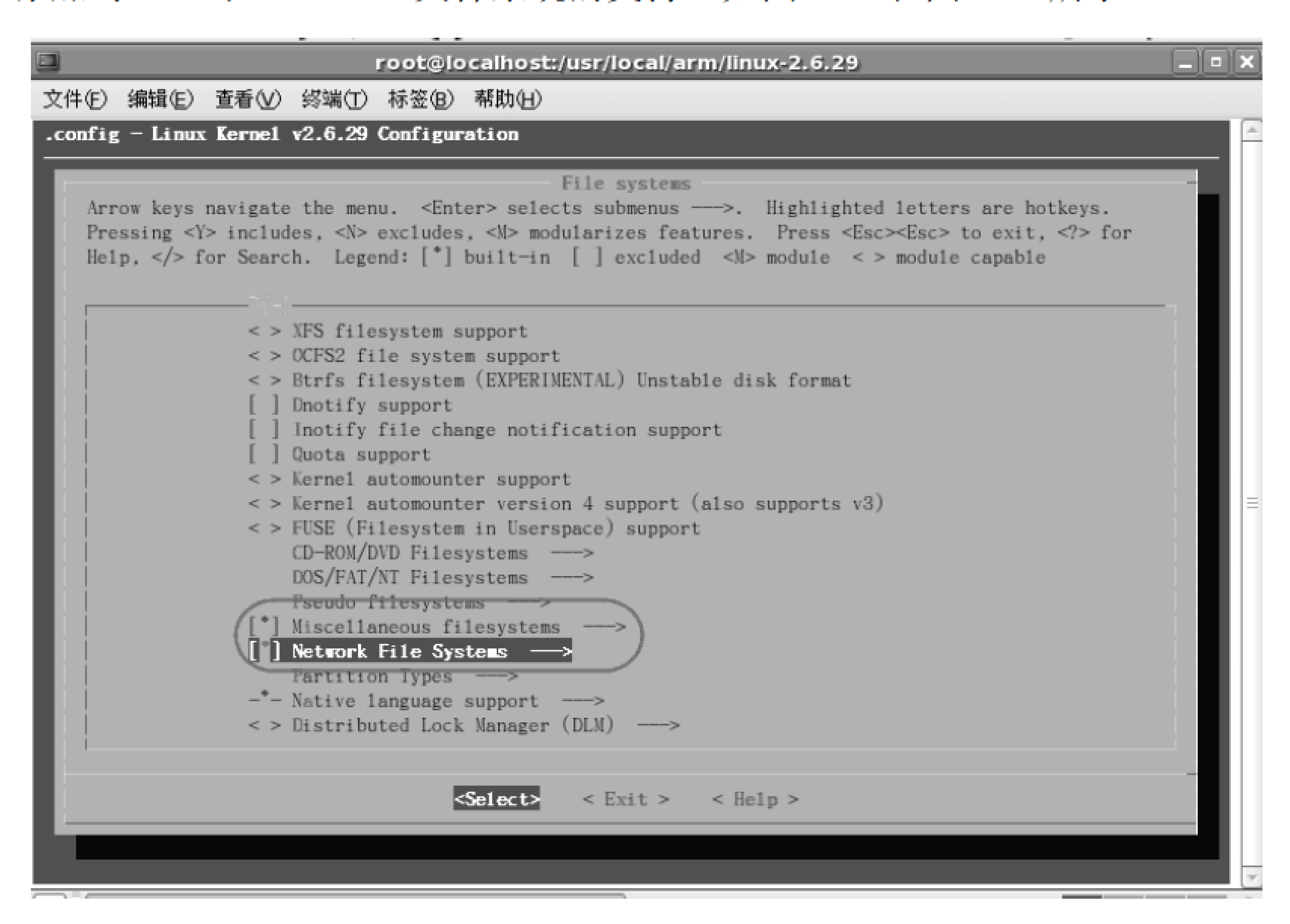



图 4.21　对 NFS 的支持

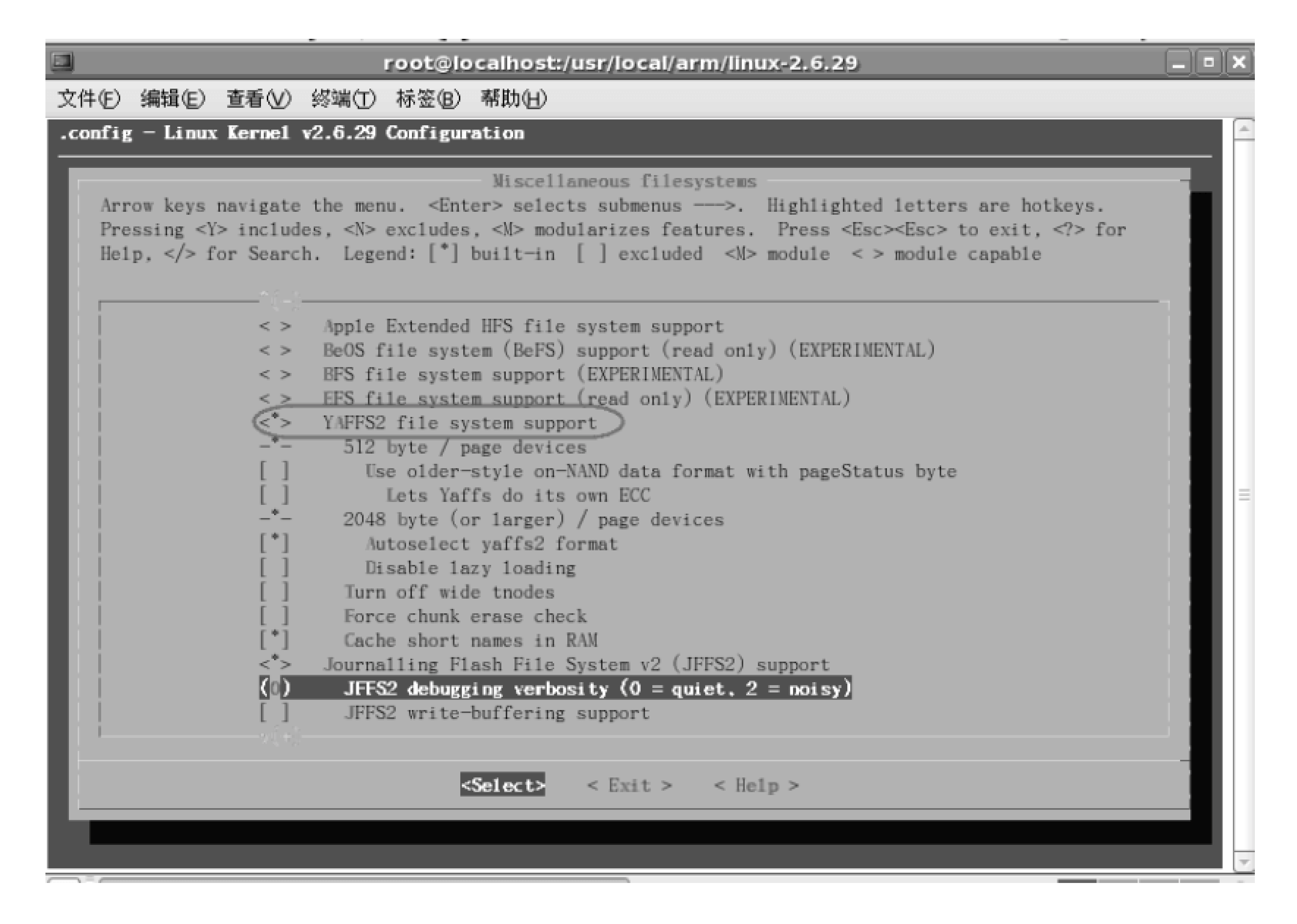



图 4.22　对 YAFFS2 的支持

如果开发板在挂载其他存储设备时，这些存储设备还包含中文时，为了正确挂载这些设备，则应该在 Native language support 中添加对字符编码的设置，如图 4.23 所示为支持简体中文的配置。

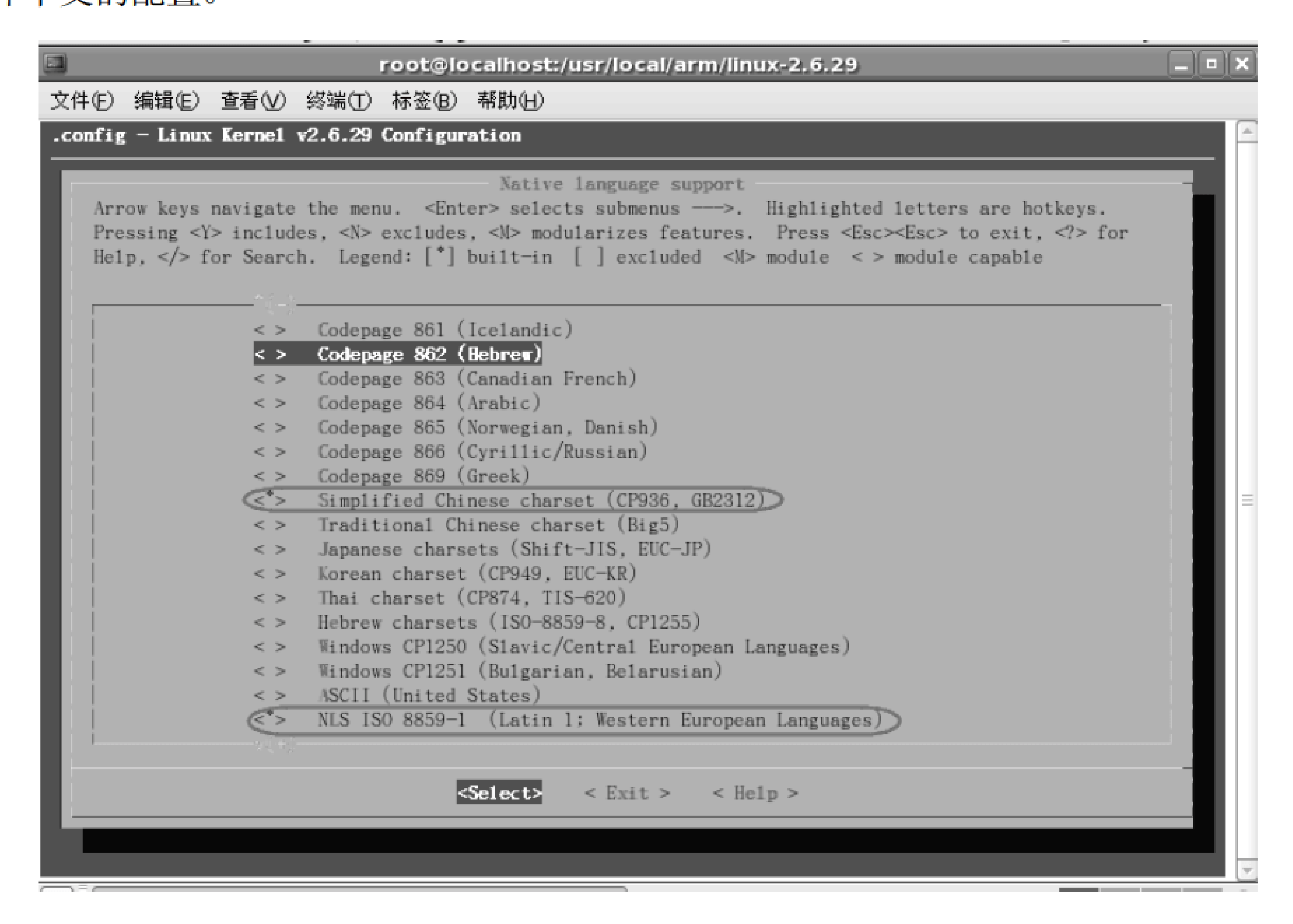



图 4.23　对字符编码的支持

（15）剩下的内核选项一般不作配置，退出内核的配置界面并保存配置。

## 4.3.4　编译内核

如果是第一次编译内核就不用清理以前的映像文件。否则可以使用 make clean 命令清理以前编译的结果。在 linux-2.6.29 目录下使用 make dep 和 make zImage 命令生成内核映像文件，编译的过程如图 4.24 所示。

```
make clean
make dep
make zImage
```

**注意**：make dep 是当程序之间有依赖关系的时候，程序发生更新时，依赖的程序会自动更新。

如果编译成功，最后会打印生成内核映像文件 zImage 及其目录。

```
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy.gz
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
```

```
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

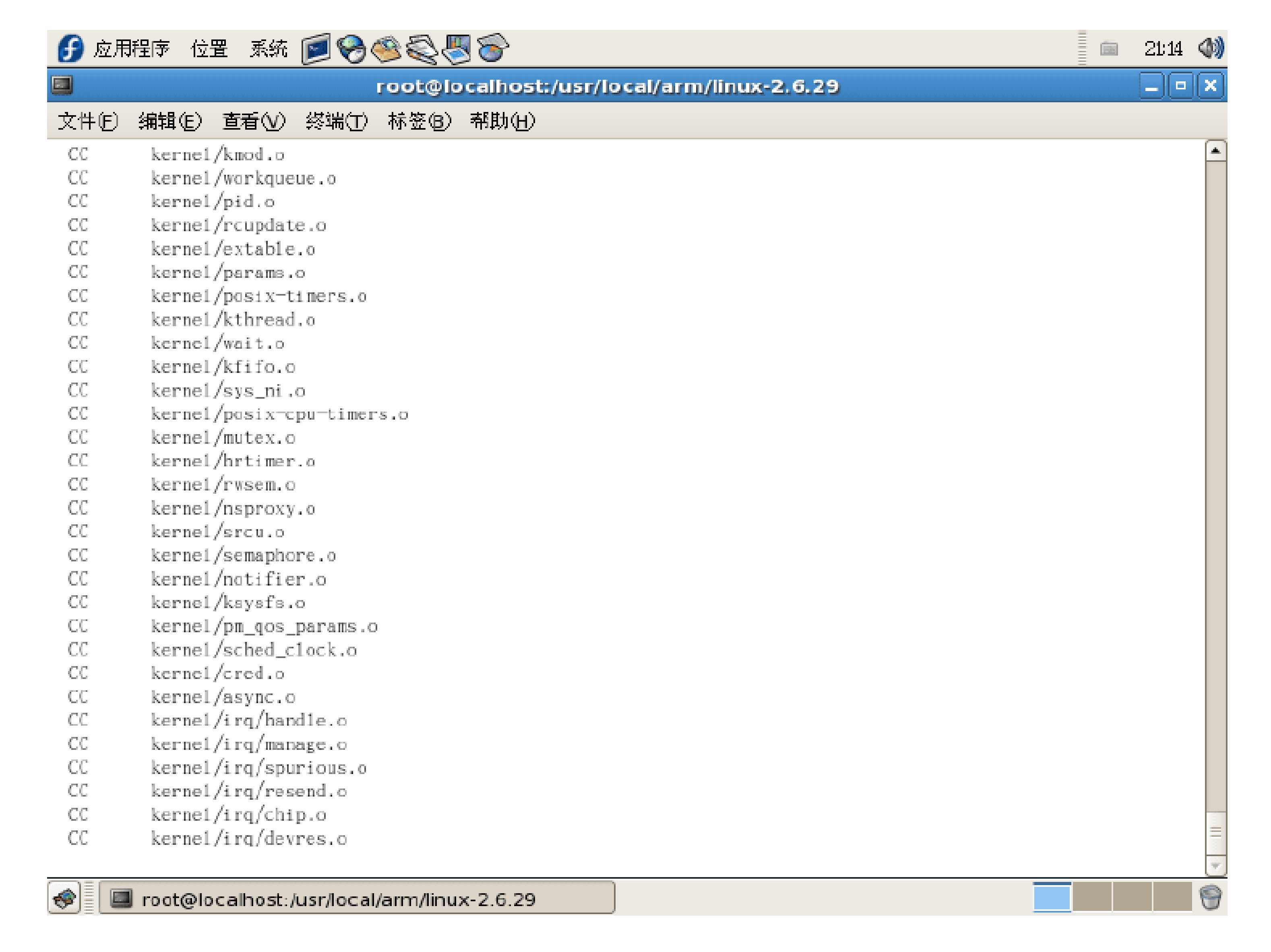



图 4.24　编译过程

# 4.4　内核映像文件移植到 ARM 板

4.3 节中，介绍了编译内核映像文件。本节中介绍将映像文件下载到 S3C2440 开发板上。如果开发板没有烧写 Bootloader，或者上位机没有安装下载映像文件工具 DNW，请参考前面的相关章节。在这里依然可以暂时使用厂家自带的文件系统。等后面讲定制文件系统后，就可以使用自制的文件系统。

## 4.4.1　移植准备

将 4.3 节生成的映像文件复制到 Windows 目录下，将要下载的文件系统放映像文件、内核映像文件放在一起，便于下载。

（1）将开发板与上位机正确连接，确定开发板电源已经插上，且开发板处于关闭状态；串口线已经正确连接；USB 线未连接。运行 DNW 工具，此时 DNW 的 COM 和 USB 状态如图 4.25 所示。

（2）确定上位机与开发板相连的串口编号。这里用的是笔记本，没有串口，采用 USB 转串口。在 Windows 设备管理器下可以看到与开发板相连的串口为 COM4，如图 4.26 所示。

（3）选择 Configuration | Options 命令，进入串口配置界面，将波特率设置为 115200，

COM Port 为设置 COM4，下载地址设置为 0x32000000，如图 4.27 所示。配置完成后单击 OK 按钮保存配置。

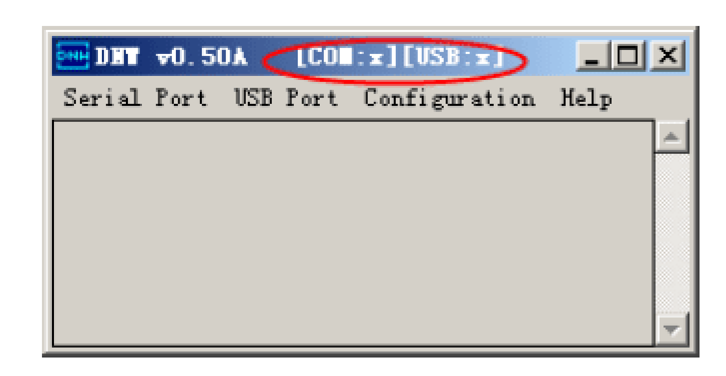



图 4.25　未连接前 DNW 状态

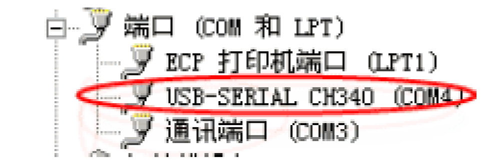



图 4.26　确定与开发板相连的串口

（4）选择 Serial Port | connect 命令，DNW 状态应该变成如图 4.28 所示的状态。

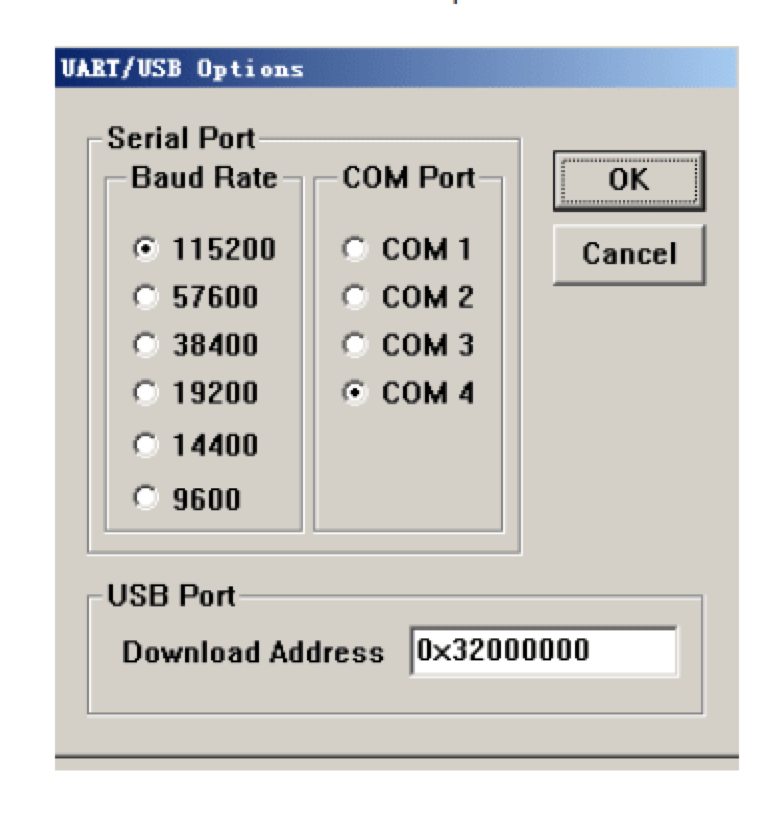



图 4.27　确定与开发板相连的串口

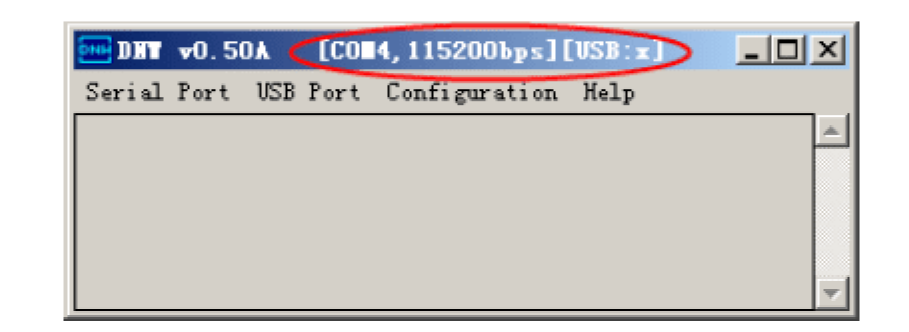



图 4.28　串口配置正确后状态

（5）确定以上步骤正确后，通过 USB 线将上位机和开发板连接起来。按住上位机的空格键，启动开发板。如果是第一次采用 USB 下载系统将会提示安装驱动。根据提示安装完驱动之后，DNW 将进入 vivi 模式。此时 USB 状态为 OK，在 DNW 显示正确的 vivi 信息，如图 4.29 所示。

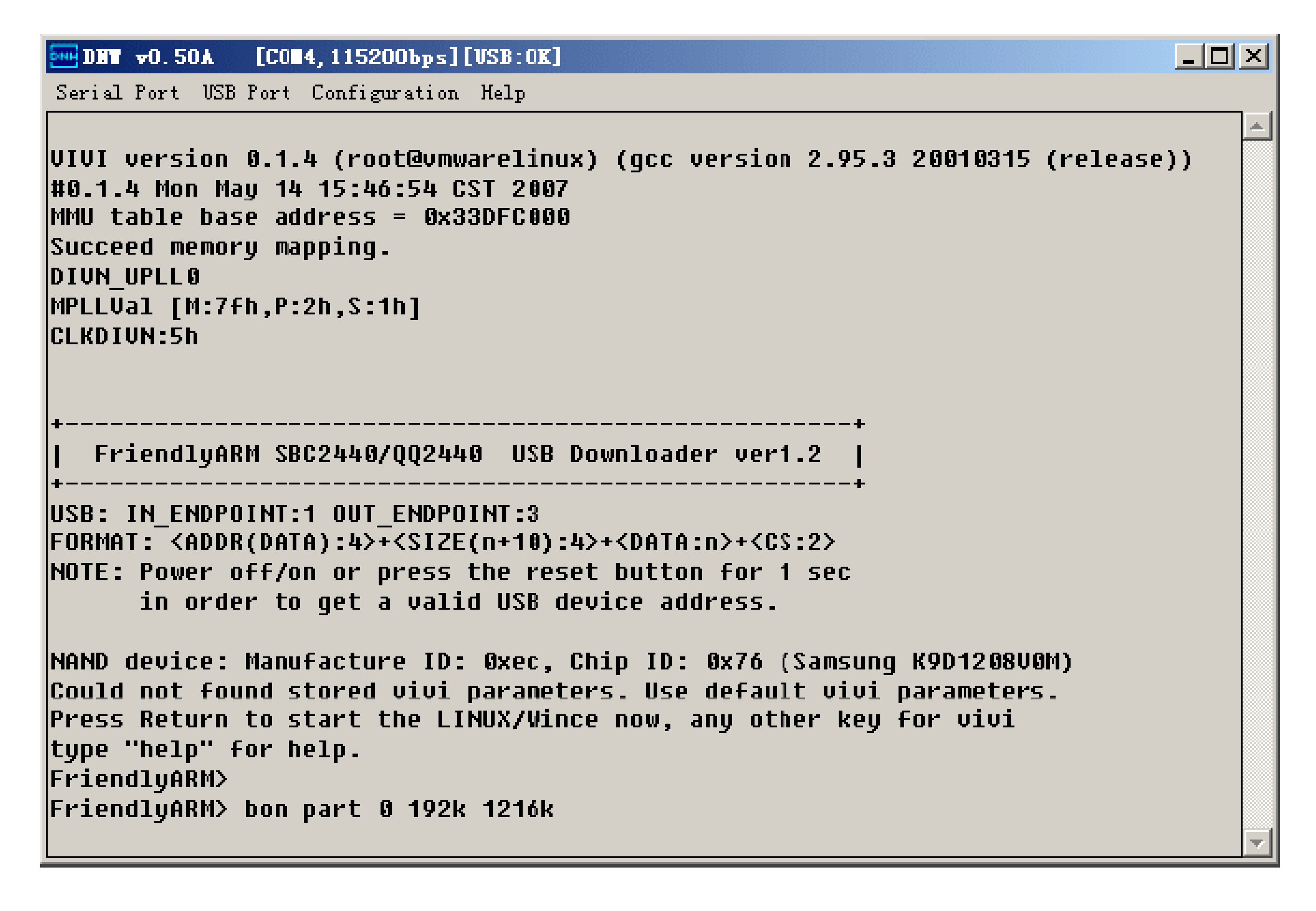



图 4.29　进入 vivi 模式

## 4.4.2　烧写系统

烧写 Linux 系统的整个过程包括格式化 Nand Flash、烧写 Bootloader、烧写内核映像文件和烧写文件系统映像文件。下面具体介绍每个步骤的详细过程。

### 1．分区格式化Flash

在 vivi 模式下输入：bon part 0 192k 1216k，将 Nand Flash 分成三个区。三个区的大小如下所示。

- 0～192k：大小为 192k；
- 192k～1216k：大小为 1M；
- 1216k～64M：大小为 63M。

执行分区命令后，会在屏幕上打印下列信息：

```
FriendlyARM> bon part 0 192k 1216k
doing partition
size = 0
size = 196608
size = 1245184
```

以上信息显示分区的起始地址。

```
check bad block
part = 0 end = 196608
part = 1 end = 1245184
part = 2 end = 67108864
part0:
    offset = 0
    size = 196608
    bad block = 0
part1:
    offset = 196608
    size = 1048576
    bad block = 0
part2:
    offset = 1245184
    size = 65847296
    bad_block = 0
```

以上信息详细打印了分区大小、坏区大小和分区的起始地址等信息。

注意：分区后不能掉电或者关电，因为此时 Nand Flash 中已经被清空。否则需要按照第 3 章介绍的方法使用 H-JTAG 重新烧写 Bootloader。

### 2．烧写Bootloader

在 vivi 模式下输入 load flash vivi u 命令。DNW 进入等待下载状态后，选择 DNW 菜单栏的 USB Post | Transmit 命令，选择 vivi.bin 文件。烧写完成后会打印如下信息。

```
FriendlyARM> load flash vivi u
USB host is connected. Waiting a download.
```

```
Now, Downloading [ADDRESS:30000000h,TOTAL:105154]
RECEIVED FILE SIZE:  105154 (102KB/S, 1S)
Downloaded file at 0x30000000, size = 105144 bytes
Found block size = 0x0001c000
Erasing...    ... done
Writing...    ... done
Written 105144 bytes
```

如果烧写成功，就会打印 Writing… …done。

🔔**注意：**在打印 USB host is connected. Waiting a download 信息后，单击 DNW 菜单栏的 USB Post|Transmit 命令。出现选择文件对话框，选择文件后开始烧写 Bootloader。

### 3．下载Linux内核文件

在 vivi 模式下输入 load flash kernel u 命令，DNW 进入等待下载状态后，单击 DNW 菜单栏的 USB Post | Transmit 命令，选择 4.4.1 节生成的内核文件 zImage，如图 4.30 所示。下载内核的过程如图 4.31 所示。

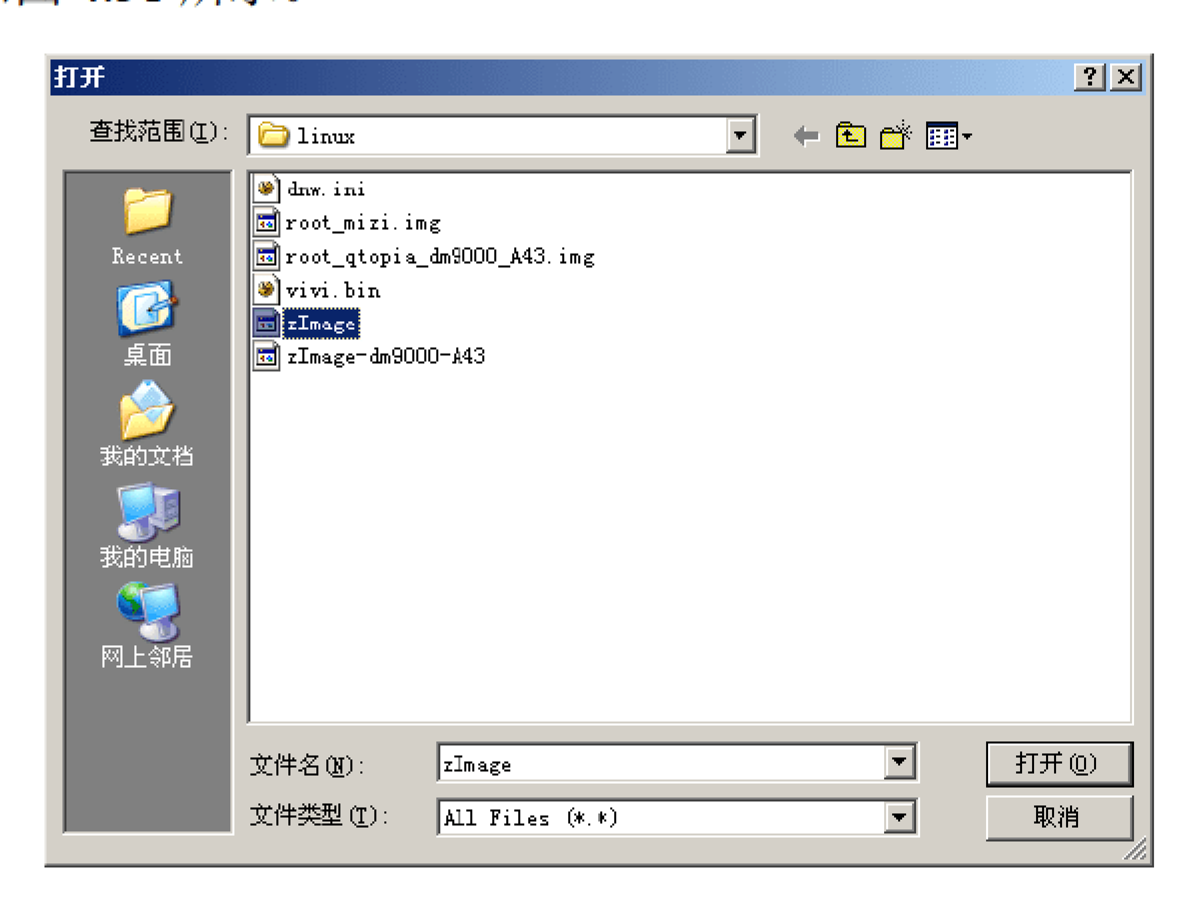



图 4.30　选择内核文件

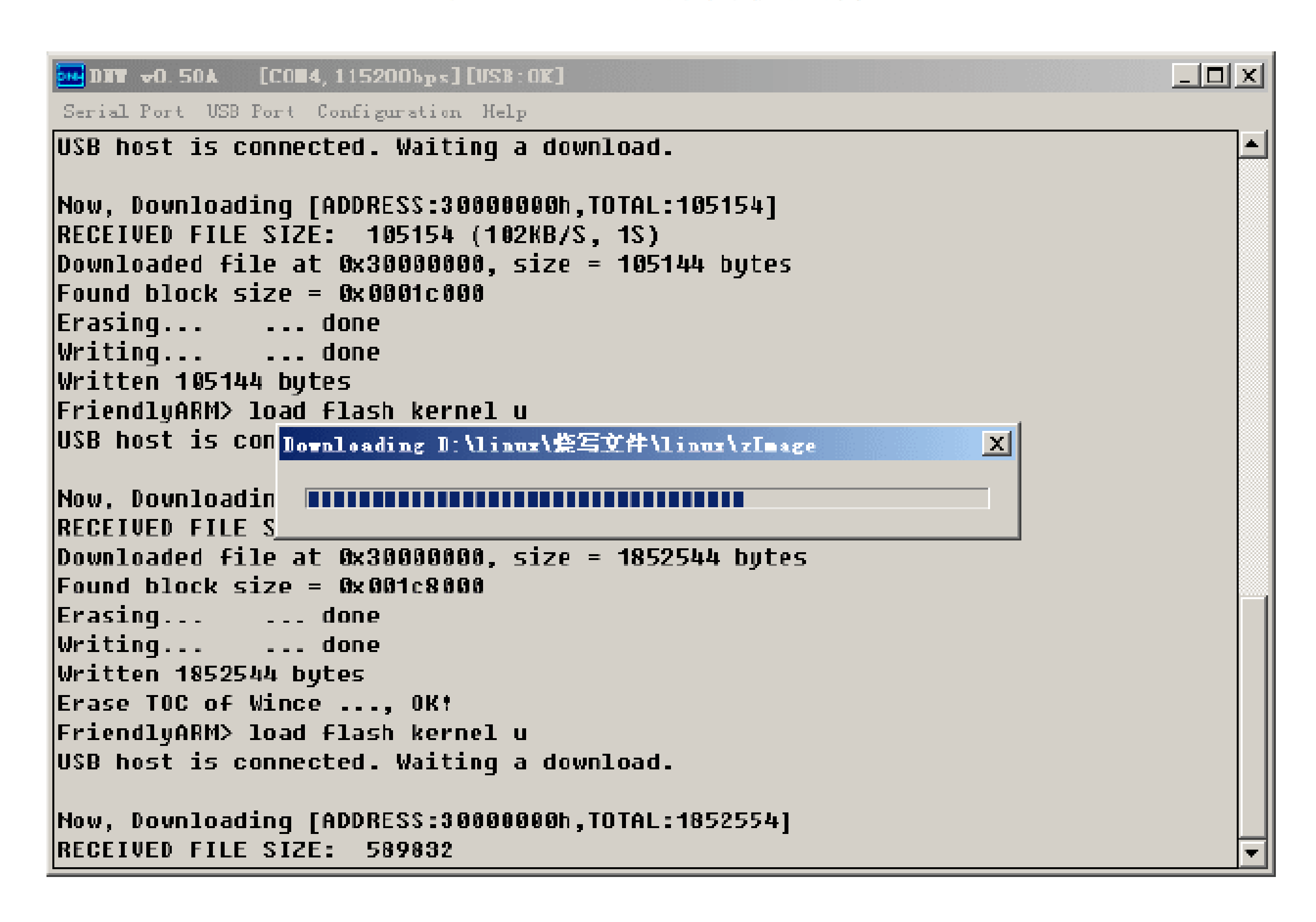



图 4.31　下载内核过程

正确下载内核完成信息如下：

```
FriendlyARM> load flash kernel u
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30000000h,TOTAL:1852554]
RECEIVED FILE SIZE: 1852554 (904KB/S, 2S)
Downloaded file at 0x30000000, size = 1852544 bytes
Found block size = 0x001c8000
Erasing...    ... done
Writing...    ... done
Written 1852544 bytes
```

注意：在打印 USB host is connected. Waiting a download 信息后，单击 DNW 菜单栏的 USB Post | Transmit 命令。

### 4．安装文件系统

接上一步操作，输入命令 loadyaffs root u 安装文件系统，暂时使用开发板厂家提供的示例文件系统。选择文件系统映像文件 root_qtopia_dm9000A43.img，如图 4.32 所示。安装文件系统过程如图 4.33 所示。

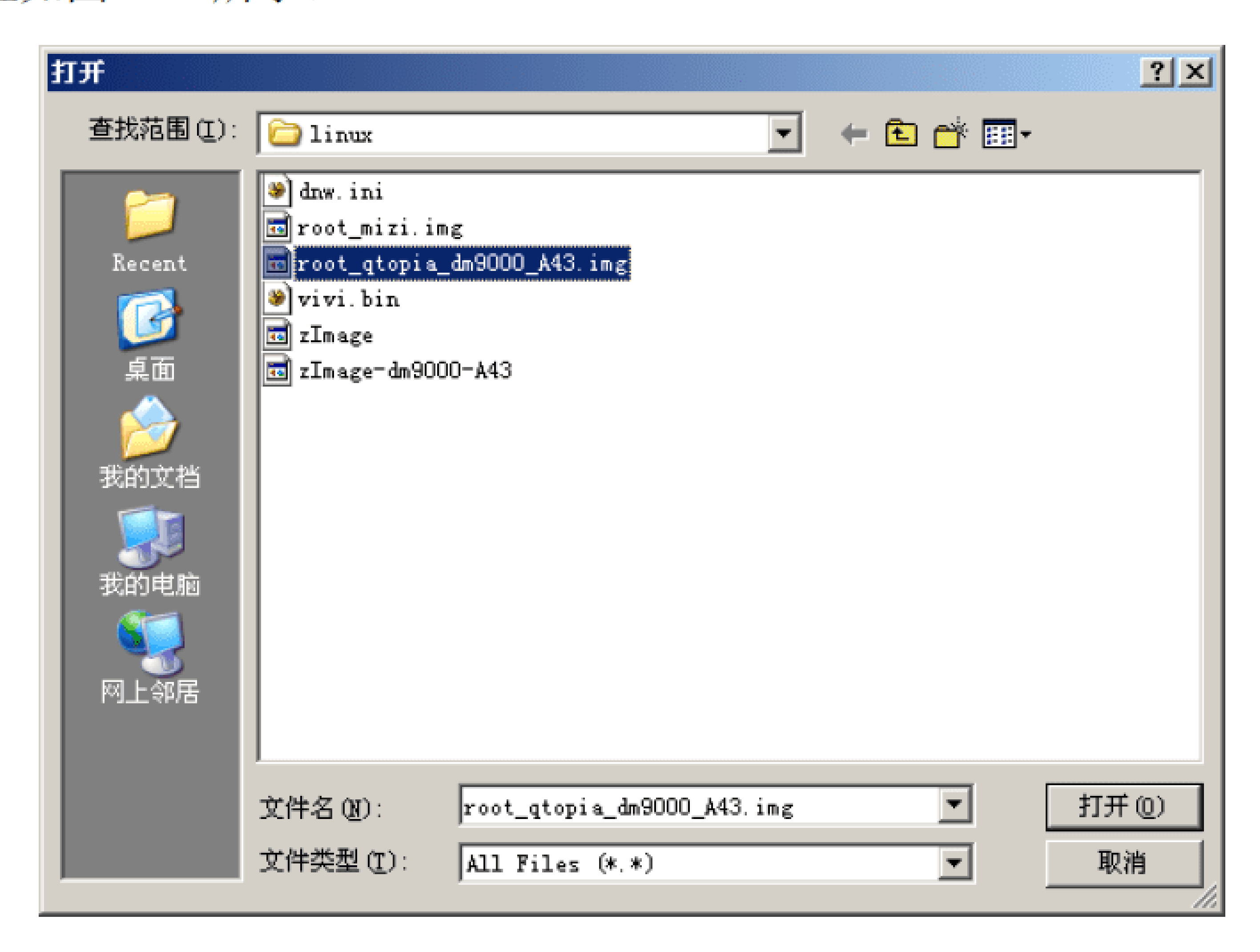



图 4.32　选择文件系统映像文件

正确安装文件系统后显示载入 yaffs 文件系统成功及文件系统的大小，打印如下信息。

```
Load yaffs OK:
Blocks scanned: 3947, Blocks erased: 3947, Blocks are bad: 0
RECEIVED and Writed FILE SIZE:45779722 (363KB/S, 123S)
```

### 5．启动系统

在 vivi 模式下输入 boot 启动系统，正确进入系统后显示如图 4.34 所示。或者直接重启开发板进入 Linux 系统。

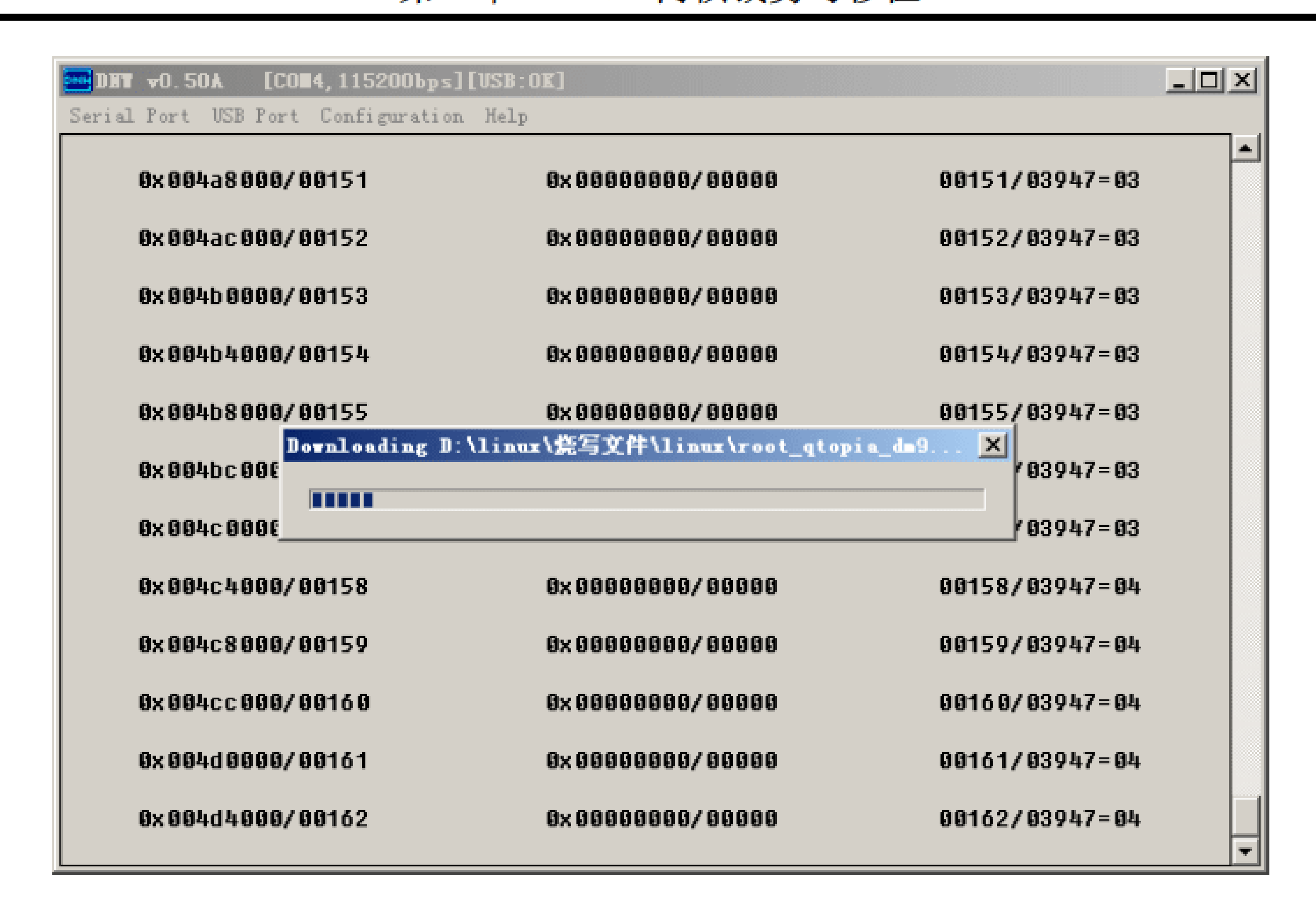



图 4.33　安装文件系统过程

```
DNW v0.50A    [COM4,115200bps][USB:x]
Serial Port  USB Port  Configuration  Help
FAT: utf8 is not a recommended IO charset for FAT filesystems, filesystem will
be case sensitive!
yaffs: dev is 32505858 name is "mtdblock2"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.2, "mtdblock2"
usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: New USB device found, idVendor=058f, idProduct=9254
usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1: Product: Generic USB Hub
usb 1-1: Manufacturer: ALCOR
usb 1-1: configuration #1 chosen from 1 choice
hub 1-1:1.0: USB hub found
hub 1-1:1.0: 4 ports detected
yaffs_read_super: isCheckpointed 0
VFS: Mounted root (yaffs filesystem) on device 31:2.
Freeing init memory: 228K
[21/Dec/2009:12:50:47 +0000] boa: server version Boa/0.94.13
[21/Dec/2009:12:50:47 +0000] boa: server built Mar 26 2009 at 15:28:42.
[21/Dec/2009:12:50:47 +0000] boa: starting server pid=485, port 80

Try to bring eth0 interface up......eth0: link down
Done

Please press Enter to activate this console.
[root@FriendlyARM /]#
```

图 4.34　正确进入系统信息

# 4.5　内 核 升 级

系统移植还包括内核升级。当开发板提供的内核和编译器版本太低，不能兼容很多新的驱动和功能时，此时就要着手考虑升级内核。本节将以 at91rm9200 为例，介绍为开发板移植高版本的内核。

## 4.5.1　准备升级内核文件

开发板自带的内核版本为 Linux-2.4.27，编译器版本为 2.95.3。在开发一些新的应用程

序和驱动时，编译器和内核不支持新的功能。准备将内核升级到 2.6 版本，编译器选择的版本为 3.4.1。需要准备的资源文件列表如下所示。

- 内核：linux-2.6.10.tar.gz；
- 针对 at91 的内核补丁：2.6.10-at91.patch.gz；
- 交叉编译器：cross-3.4.1.tar.bz2。

## 4.5.2　移植过程

下面详细介绍一下移植过程。

（1）将所有文件复制到工作目录下，然后解压内核文件和编译器文件。

（2）为内核打补丁。

```
#cd linu-2.6.10
#patch -p1< 2.6.10-at91.patch.gz
```

（3）修改 Makefile，修改编译环境。

```
ARCH = arm
CROSS_COMPILE =/usr/local/arm/3.4.1/bin/arm-linux-
```

（4）修改 machine ID。如果这一步省略，会在移植到开发板后 Bootloader 引导时出现机器 ID 错误的现象。出错的 ID 号将以十六进制给出，将其转化为十进制，替换 mach-types 文件中的对应项。这里移植后报的错误是 0xFB，即对应十进制 251。

```
#vi  /usr/local/arm/linux-2.6.10/arch/arm/tools/mach-types
```

找到

```
at91rm9200dk          ARCH_AT91RM9200DK      AT91RM9200DK          262
```

将其修改为：

```
at91rm9200dk          ARCH_AT91RM9200DK      AT91RM9200DK          251
```

（5）制作 uImage 文件。在内核目录下建议一个名为 mkimage 的文件，其内容如下：

```
/usr/local/arm/3.4.1/bin/arm-linux-objcopy -O binary -S vmlinux linux.bin
gzip -v9 linux.bin
./mkimage -A arm -O linux -T kernel -C gzip -a 0x20008000 -e 0x20008000 -d
linux.bin.gz uImage
```

（6）对内核进行配置。执行 make at91rm9200dk_defconfig 实际上就是完成对内核的配置。

```
#make at91rm9200dk_defconfig
```

其具体配置如下：

```
* Plug and Play support
* Block devices
RAM disk support (BLK DEV RAM) [Y/n/m/?] y
  Default number of RAM disks (BLK DEV RAM COUNT) [16] 16
  Default RAM disk size (kbytes) (BLK_DEV_RAM_SIZE) [8192] 8192
  Initial RAM disk (initrd) support (BLK_DEV_INITRD) [Y/n/?] y
```

```
Source directory of cpio_list (INITRAMFS_SOURCE) []
Packet writing on CD/DVD media (CDROM_PKTCDVD) [N/m/y/?] n
* IO Schedulers
Anticipatory I/O scheduler (IOSCHED_AS) [Y/n/m/?] y
* Multi-device support (RAID and LVM)
* Networking support
Networking support (NET) [Y/n/?] y
  * Networking options
  Packet socket (PACKET) [Y/n/m/?] y
  Unix domain sockets (UNIX) [Y/n/m/?] y
    TCP/IP networking (INET) [Y/n/?] y
        IP: kernel level autoconfiguration (IP_PNP) [Y/n/?] y
      IP: BOOTP support (IP_PNP_BOOTP) [Y/n/?] y
  IP: TCP socket monitoring interface (IP_TCPDIAG) [Y/n/m/?] y
  * Network packet filtering (replaces ipchains)
  * SCTP Configuration (EXPERIMENTAL)
  * QoS and/or fair queueing
  * Network testing
* Amateur Radio support
* IrDA (infrared) subsystem support
* Bluetooth subsystem support
Network device support (NETDEVICES) [Y/n/?] y
  * Ethernet (10 or 100Mbit)
  Ethernet (10 or 100Mbit) (NET_ETHERNET) [Y/n/?] y
    Generic Media Independent Interface device support (MII) [Y/?] y
    AT91RM9200 Ethernet support (ARM_AT91_ETHER) [Y/n/m/?] y
      RMII interface (ARM_AT91_ETHER_RMII) [Y/n/?] y
  * Ethernet (1000 Mbit)
  * Ethernet (10000 Mbit)
  * Token Ring devices
  * Wireless LAN (non-hamradio)
  * Wan interfaces
* SCSI device support
* Fusion MPT device support
* IEEE 1394 (FireWire) support
* I2O device support
* ISDN subsystem
* Input device support
* Userland interfaces
Mouse interface (INPUT_MOUSEDEV) [Y/?] (NEW) y
  Horizontal screen resolution (INPUT_MOUSEDEV_SCREEN_X) [1024] 1024
  Vertical screen resolution (INPUT_MOUSEDEV_SCREEN_Y) [768] 768
* Input I/O drivers
* Input Device Drivers
* Character devices
* Serial drivers
* Non-8250 serial port support
AT91RM9200 serial port support (SERIAL_AT91) [Y/n/m/?] y
  Support for console on AT91RM9200 serial port (SERIAL_AT91_CONSOLE) [Y/n/?]
y
Legacy (BSD) PTY support (LEGACY_PTYS) [Y/n/?] y
  Maximum number of legacy PTY in use (LEGACY_PTY_COUNT) [256] 256
* IPMI
* Watchdog Cards
Watchdog Timer Support (WATCHDOG) [Y/n/?] y
  Disable watchdog shutdown on close (WATCHDOG_NOWAYOUT) [Y/n/?] y
  * Watchdog Device Drivers
  AT91RM9200 watchdog (AT91_WATCHDOG) [Y/n/m/?] y
   * USB-based Watchdog Cards
```

```
* Ftape, the floppy tape device driver
SPI driver for AT91 processors (AT91 SPI) [Y/n/?] y
  SPI device interface for AT91 processors (AT91 SPIDEV) [Y/n/?] y
* I2C support
I2C support (I2C) [Y/n/m/?] y
  I2C device interface (I2C_CHARDEV) [Y/n/m/?] y
  * I2C Algorithms
  * I2C Hardware Bus support
  Atmel AT91RM9200 I2C Two-Wire interface (TWI) (I2C_AT91) [Y/n/m/?] y
  * Hardware Sensors Chip support
  * Other I2C Chip support
* Multimedia devices
* Digital Video Broadcasting Devices
* File systems
Second extended fs support (EXT2_FS) [Y/n/m/?] y
* CD-ROM/DVD Filesystems
* Pseudo filesystems
/proc file system support (PROC_FS) [Y/n/?] y
/dev file system support (OBSOLETE) (DEVFS_FS) [Y/n/?] y
  Automatically mount at boot (DEVFS MOUNT) [Y/n/?] y
  Debug devfs (DEVFS_DEBUG) [N/y/?] n
Virtual memory file system support (former shm fs) (TMPFS) [Y/n/?] y
* Miscellaneous filesystems
Compressed ROM file system support (cramfs) (CRAMFS) [Y/n/m/?] y
* Network File Systems
* Partition Types
* Native Language Support
* Profiling support
* Graphics support
* Console display driver support
* Sound
* Misc devices
* USB support
Support for Host-side USB (USB) [Y/n/m/?] y
  USB verbose debug messages (USB DEBUG) [Y/n/?] y
  * Miscellaneous USB options
* USB Host Controller Drivers
SL811HS HCD support (USB_SL811_HCD) [N/m/y/?] n
* USB Device Class drivers
USB Mass Storage support (USB STORAGE) [N/m/y/?] n
* USB Input Devices
  * USB HID Boot Protocol drivers
* USB Imaging devices
* USB Multimedia devices
* Video4Linux support is needed for USB Multimedia device support
* USB Network Adapters
* USB port drivers
* USB Serial Converter support
* USB Miscellaneous drivers
* USB ATM/DSL drivers
* USB Gadget Support
* MMC/SD Card support
* Kernel hacking
Kernel debugging (DEBUG_KERNEL) [Y/n/?] y
* Security options
* Cryptographic options
* Library routines
CRC32 functions (CRC32) [Y/?] y
```

上面已经对内核做了详细的配置，考虑到内容比较多的情况，省略了没有配置的选项。可以通过 make menuconfig 去查看对 System Type（系统类型）的修改情况以确认进行正确的配置，如图 4.35 所示。

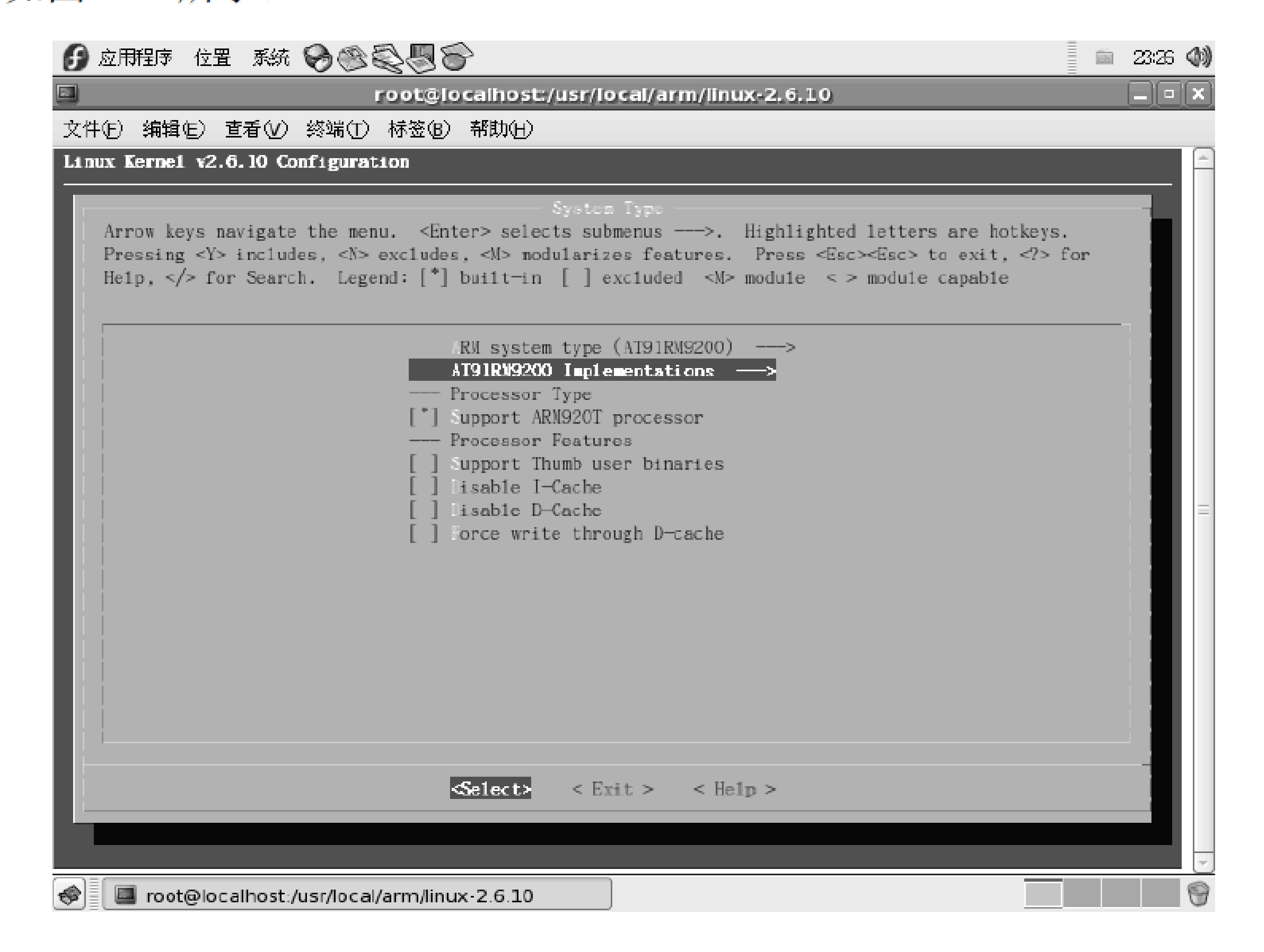



图 4.35　系统类型已经被设置为 AT91RM9200

（7）编译内核生成映像文件。

```
#make clean
#make dep
.#/mkimage
```

## 4.6　小　　结

本章主要讲解 Linux 内核的目录结构、Linux 内核配置选项及裁剪内核、编译内核。最后结合实例讲解内核移植和内核升级的具体过程。在开始接触内核移植时，不提倡初学者拿到源码就直接进行裁剪配置，这样经常会由于忽略了某个选项导致移植的时候失败。最好的办法是首先导入内核自带的配置，在这些配置的基础上根据自己的需要进行裁剪。

# 第 5 章　嵌入式文件系统制作

Linux 支持多种文件系统，包括 Ext2、Ext3、Vfat、Ntfs、Iso9660、Jffs、Romfs 和 Nfs 等。为了对各种不同类型的文件系统进行统一管理，Linux 引入了虚拟文件系统 VFS（Virtual File System），为各类文件系统提供一个统一的操作界面和应用编程接口。本章主要介绍各种嵌入式文件系统的特点和嵌入式文件系统的制作过程。嵌入式文件系统包括 Ramdisk、Jffs2、Yaffs、Cramfs、Romfs 和 Ramfs/Tmpfs。了解各种嵌入式文件系统的特点，有利于读者选择适合自己硬件条件的文件系统。

## 5.1　文件系统选择

在进行嵌入式系统开发过程中，文件系统的选择和制作与硬件条件息息相关。根据硬件（Flash 或 RAM）的特性来指定相应的文件系统，能够充分利用硬件资源及提高系统效率。因为目前大部分的嵌入式文件系统都是建立在 Flash 之上，下面将介绍 Flash 硬件方案比较与 Flash 的特点。

### 5.1.1　Flash 硬件方案比较

Flash（闪存）是嵌入式系统的主要存储介质，其特点为写入操作只能把对应位置的 1 修改为 0，而不能把 0 修改为 1。因此，对于 Flash 的擦除操作是把对应存储块的内容恢复为 1。一般情况下，向 Flash 写入内容时，首先必须擦除对应的存储区间，擦除是以块（block）为单位进行。闪存技术主要有 NOR 和 NAND 两种技术。Flash 存储器的擦写次数是有限的，NAND 闪存设备有特殊的硬件接口和读写时序。因此，必须根据 Flash 硬件特性设计符合应用要求的文件系统。下面介绍选择硬件方案的原则。

硬件方案的总体原则是：用于数据存储采用 NAND Flash，用于代码存储采用 NOR Flash。依据这一总则，系统架构师或者硬件设计师在硬件选型阶段可以灵活地将两种闪存结合使用，用 NOR Flash 存放引导程序和根文件系统，用 NAND Flash 存放用户文件系统，使两种闪存进行优势互补。目前在手机、PocketPC、PDA 及电子词典等设备的设计中基本采用类似的方案。

在选择存储解决方案时，为了获得最高的性价比，设计师在速度、存储密度、成本、开发周期等多种因素之间进行权衡。以手机的存储解决方案为例：NOR Flash 采用支持 XIP（eXecute In Place，芯片内执行）技术能够直接运行操作系统，速度快，既简化了设计，又降低了成本，所以许多手机硬件方案都采用 NOR Flash＋RAM。

NOR Flash 的缺点是存储密度较低，为了提高手机的存储容量也有方案采用 NAND

Flash＋RAM。而同时追求速度和容量的手机硬件方案则会采用 NOR Flash＋NAND Flash＋RAM。使用 NAND Flash 的技术难度超过 NOR Flash，几乎每个 NAND 器件都存在坏的扇区，需要纠错码来维持数据。而且，在 NAND 器件上运行代码，需要存储技术驱动程序 MTD（Memory Technology Device）技术的支持。

在表 5.1 中给出了两种 Flash 的特性进行对比，在具体的硬件选型阶段读者可以参考该表进行具体的硬件选型。

表 5.1　NOR Flash与NAND Flash比较

| 磁 盘 类 型 | NOR Flash | NAND Flash |
|---|---|---|
| 使用难易程度 | 接口时序同 SRAM，易使用 | 地址/数据线复用，数据位较窄 |
| 读速度 | 读取速度比较快 | 读取速度比较慢 |
| 擦除速度 | 擦除速度慢，以 64～128KB 的块为单位 | 写（编程）和擦除操作的速率快，以 8～32KB 的块为单位 |
| 写速度 | 写入速度慢（因为一般要先擦除） | 写入速度快 |
| 应用场合 | 随机存取速度较快，支持 XIP（eXecute In Place，芯片内执行），适用于代码存储。在嵌入式系统中，常用于存放引导程序、根文件系统等 | 顺序读取速度较快，随机存取速度慢，适用于数据存储（如大容量的多媒体应用）。用于嵌入式系统中时，通常存放用户文件系统等 |
| 存储密度 | 单片容量较小，1～32MB | 单片容量较大，8～128MB，提高了单元密度 |
| 使用成本 | 成本高 | 成本低 |
| 使用寿命 | NOR 的擦写次数为十万次 | NAND 闪存中每个块的最大擦写次数为一百万次 |
| 软件支持 | 在 NOR 器件上运行代码时不需要其他驱动程序支持 | 在 NAND 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序（MTD），NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD 驱动程序支持 |

### 5.1.2　嵌入式文件系统的分层结构

不同类型的文件系统具有不同的特点，根据系统需求、采用的存储设备的硬件特性等，采用不同的文件系统或者文件系统组合。在嵌入式 Linux 应用中，主要的存储设备为 RAM（DRAM，SDRAM）和 ROM（常采用 Flash 存储器），常见的基于存储设备的文件系统类型包括 Ramdisk、JFFS2、YAFFS、Cramfs、Romfs 和 Ramfs/Tmpfs 等。下面给出 Linux 系统下文件系统的分层结构，如图 5.1 所示。

从图 5.1 中可以看出，嵌入式文件系统主要有基于 Flash 的文件系统和基于 RAM 的文件系统，接下来将分别基于这两种硬件的文件系统特点和原理进行介绍。

## 5.2　基于 Flash 的文件系统

基于 Flash 的文件系统主要包括 JFFS2、YAFFS、Cramfs 和 Romfs 等。各种文件系统

具有不同的特点，下面分别进行介绍。

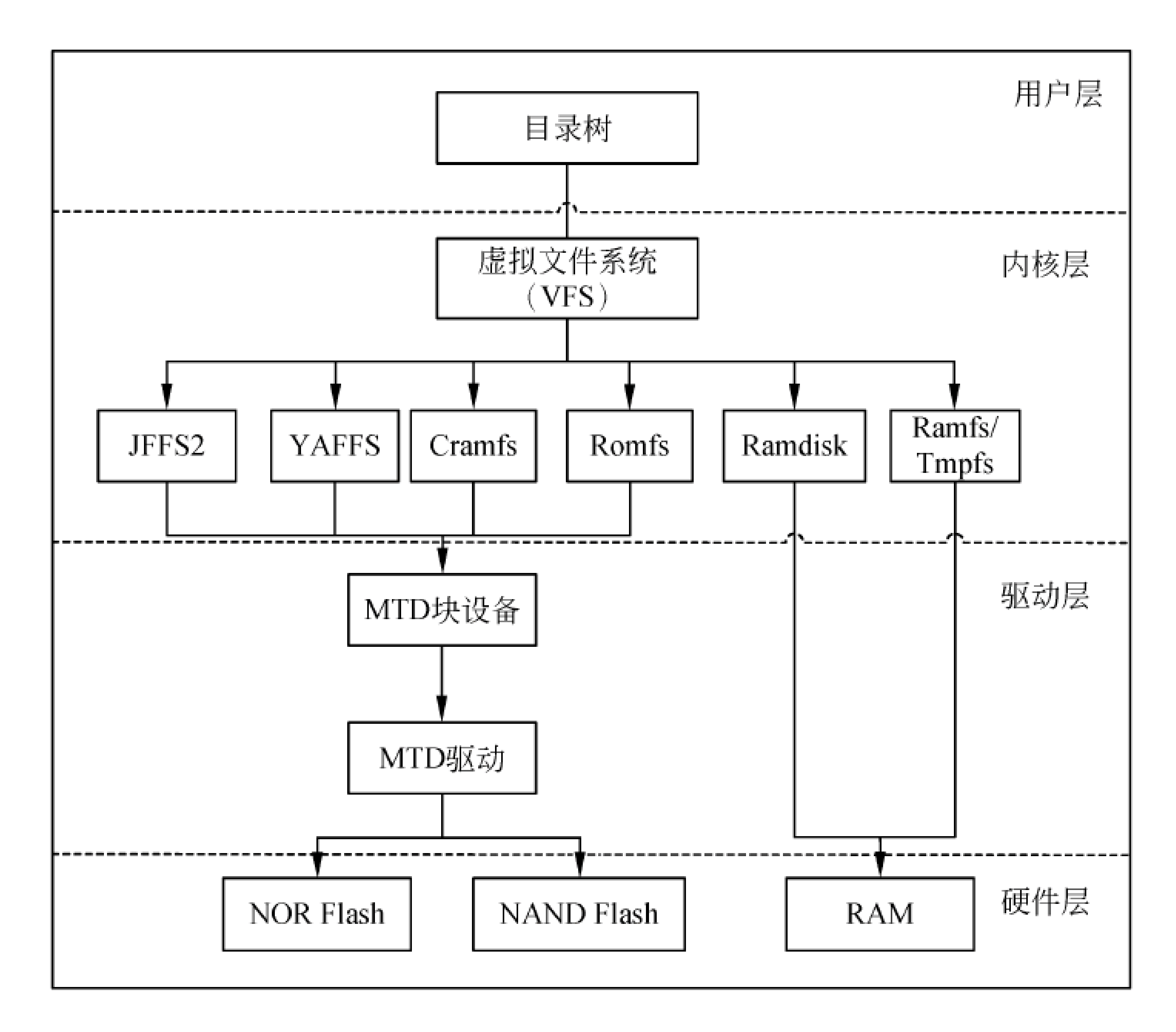



图 5.1　文件系统的分层结构

## 5.2.1　JFFS 文件系统（Journalling Flash FileSystem）

JFFS 系列日志文件系统包括 JFFS1、JFFS2 和 JFFS3，JFFS3 正在开发中，JFFS2 比 JFFS1 有很多改进的地方，所以目前通常使用 JFFS2。

### 1．JFFS2的工作原理

当文件系统加载时扫描整个 Flash 的内容，将信息读入日志结点 jffs2_raw_inode，然后根据该信息建立文件系统。修改操作是先分配新结点 jffs2_raw_inode，将内容写入新结点，然后将原来的结点标记为脏数据。当系统接近满或者已满时就要进行垃圾收集，需要扫描整个 Flash 中的结点，将标记为脏的结点进行回收。

```
struct jffs2_raw_inode
{
    jint16_t magic;           //固定的魔数 magic number
    jint16_t nodetype;        //结点类型设置为 JFFS2_NODETYPE_INODE
    jint32_t totlen;          //包括有效数据在内的结点总长度
    jint32_t hdr_crc;         //jffs2_unknown_node 部分的 CRC 校验
    jint32_t ino;             //结点数
    jint32_t version;         //版本数
    jmode t mode;             //文件类型
    jint16_t uid;             //文件的属主
    jint16_t gid;             //文件的属组
```

```
    jint32_t isize;            //实际长度
    jint32_t atime;            //上一次访问时间
    jint32_t mtime;            //上一次修改时间
    jint32_t ctime;            //创建时间
    jint32_t offset;           //结点对应的数据在文件中的起始地址
    jint32_t csize;            //压缩数据的长度
    jint32_t dsize;            //压缩后数据的有效长度
    uint8_t compr;             //当前使用的压缩算法
    uint8_t usercompr;         //用户指定的压缩算法
    jint16_t flags;            //标志位
    jint32_t data_crc;         //数据 CRC
    jint32_t node_crc;         //头结点 CRC
    uint8_t data[0];
};
```

### 2．JFFS2的主要特点

JFFS2，日志闪存文件系统版本 2（Journalling Flash FileSystem v2）。主要用于 NOR Flash，基于 MTD 驱动层。JFFS2 的主要特点如下：

- 可读写；
- 使用基于哈希表的日志结点结构，大大提高了对结点的操作速度；
- 支持数据压缩；
- 提供了“写平衡”支持；
- 支持多种结点类型（数据 I 结点、目录 I 结点等），JFFS 只支持一种结点；
- 提高了对闪存的利用率，降低了内存的消耗。

注意：“写平衡”是在垃圾收集中实现的。垃圾收集的时候会读取系统时间，通过系统时间产生一个伪随机数。使用这个伪随机数结合不同的待回收链表选择要进行回收的链表。使用写平衡策略能提供较好的写平衡效果。

JFFS2 与 JFFS1 相比，加快了对结点的操作速度；支持更多的结点类型；提高了对闪存的利用率，降低了内存的利用率。JFFS2 与 JFFS3 相比，根本区别在于 JFFS3 将索引信息放在闪存上，而 JFFS2 将索引信息放在内存上。

### 3．JFFS2的挂载过程

JFFS2 的挂载过程主要分为 4 个过程：

（1）JFFS2 扫描闪存介质，检查每个结点 jffs2_raw_inode 的 CRC 校验码是否合法，同时分配 struct jffs2_inode_cache 和 struct jffs2_raw_node_ref。

（2）扫描每个结点的物理结点链表，标识出过时的物理结点；将每个合法的 dentry 结点相应的 jffs2_inode_cache 中的字段 nlink 加 1。

（3）找到 nlink 为 0 的 jffs2_inode_cache，释放对应的结点。

（4）释放扫描过程中的临时信息。

### 4．JFFS2的优点和缺点

JFFS2 的优点有：其一，碎片收集对象是基于一个扇区而不是基于整个文件系统，删除的对象也是扇区，因此删除操作的时间短。其二，遇到坏扇区时进行标记而使用可用扇

区，延长了设备的写生命周期。

JFFS 系列文件系统存在下面的缺点：

- 文件系统已满或者接近满时，系统无法分配新的结点就必须进行垃圾收集；
- 垃圾收集就是从头开始扫描日志结点（jffs2_raw_inode）标记脏数据结点，这样使文件系统变得非常缓慢。

由于 JFFS 系列文件系统存在上述缺点，不适用于 NAND Flash。因为 NAND Flash 的容量一般较大，导致 JFFS 系列文件系统为维护日志结点所占用的内存空间迅速增大；其次，JFFS 系列文件系统在挂载时需要扫描整个 Flash 的内容，以找出所有的日志结点，建立文件结构，对于大容量的 NAND Flash 会耗费大量时间。

## 5.2.2　YAFFS 文件系统（Yet Another Flash File System）

YAFFS 文件系统包括 YAFFS 和 YAFFS2。YAFFS/YAFFS2 是专门为嵌入式系统使用 NAND Flash 而设计的一种日志型文件系统，适用于大容量的存储设备。与 JFFS 相比，它减少了一些功能（例如不支持数据压缩），所以速度更快，挂载时间较短，对内存的占用较小。

### 1．YAFFS文件系统的特点

YAFFS/YAFFS2 自带 NAND 芯片驱动，提供了嵌入式系统直接访问文件系统的 API，这样用户可以不使用 Linux 中的 MTD 与 VFS，直接对文件系统操作。当然，用户也可以通过 MTD 驱动程序来访问文件系统。

### 2．YAFFS与YAFFS2的区别

YAFFS 与 YAFFS2 的主要区别在于，YAFFS 仅支持小页（512 Bytes）NAND Flash，而 YAFFS2 能够支持大页（2KB）NAND Flash。另外，YAFFS2 在内存空间占用、垃圾回收速度、读/写速度等方面都有较大改进。

### 3．YAFFS/YAFFS2的工作原理

YAFFS/YAFFS2 根据 NAND Flash 的存取特点，将文件组织成固定大小（512 Bytes/2KB）的数据段。对文件系统上的所有内容（比如正常文件，目录，链接，设备文件等）都统一当作文件来处理，每个文件都有一个页面专门存放文件头，文件头保存了文件的模式、所有者 ID、组 ID、长度、文件名、Parent Object ID 等信息。根据 NAND Flash 的特点，NAND Flash 上的每一页数据都留有额外的空间用于存储附加信息，一般 NAND 驱动只占有该空间的一部分，YAFFS 文件系统正是利用了这部分空间中剩余的部分来存储文件系统相关的内容。为了了解 YAFFS 工作的原理，下面通过分析源码（fs/yaffs_guts.c）认识 YAFFS 是如何进行分配和删除页（代码中表示为 chunk）的。

函数 yaffs_AllocateChunk()用来从 block 中分配存储空间。其代码如下：

```
static int yaffs_AllocateChunk(yaffs_Device *dev, int useReserve,
        yaffs_BlockInfo **blockUsedPtr)
/*
dev 该指针用来记录 NAND 器件属性和使用情况，并且维护一组 NAND 操作函数指针。
```

```
useReserve 表示是否使用保留空间。
blockUsedPtr 记录 block 内还有多少空闲页信息。
*/
{
    int retVal;
    yaffs_BlockInfo *bi;
    //当 block 内的所有页都分配完时 dev->allocationBlock 的值为-1
    if (dev->allocationBlock < 0) {
        /*在下一个 block 中进行分配*/
        dev->allocationBlock = yaffs_FindBlockForAllocation(dev);
        dev->allocationPage = 0;
    }
    if (!useReserve && !yaffs_CheckSpaceForAllocation(dev)) {
        /*没有足够的空间进行分配时，就要等到允许使用保留空间*/
        return -1;
    }
    if (dev->nErasedBlocks < dev->nReservedBlocks
            && dev->allocationPage == 0) {
        T(YAFFS_TRACE_ALLOCATE, (TSTR("Allocating reserve" TENDSTR)));
    }
    /*分配到页*/
    if (dev->allocationBlock >= 0) {
        bi = yaffs_GetBlockInfo(dev, dev->allocationBlock);
                                            //获得 block 信息
        retVal = (dev->allocationBlock * dev->nChunksPerBlock) + dev->
        allocationPage;                     //计算分配的页大小
        bi->pagesInUse++;                   //使用的页数加 1
       yaffs_SetChunkBit(dev, dev->allocationBlock, dev->allocationPage);
                                            //在分配的 block 中标记分配的页
        dev->allocationPage++;              //更新分配页数加 1
        dev->nFreeChunks--;                 //更新空闲块数减 1
        /*如果 block 已满则设置其状态为 full */
        if (dev->allocationPage >= dev->nChunksPerBlock) {
            bi->blockState = YAFFS_BLOCK_STATE_FULL;
            dev->allocationBlock = -1;
        }
        if (blockUsedPtr)
            *blockUsedPtr = bi;             //更新使用的页数
        return retVal;
    }
    T(YAFFS_TRACE_ERROR,
            (TSTR("!!!!!!!!! Allocator out !!!!!!!!!!!!!!!!!" TENDSTR)));
    return -1;
}
```

函数 yaffs_DeleteChunk()用来释放 block 中分配的空间。其代码如下：

```
void yaffs_DeleteChunk(yaffs_Device *dev, int chunkId, int markNAND, int
lyn)
/*
dev 该指针用来记录 NAND 器件属性和使用情况，并且维护一组 NAND 操作函数指针。
chunkId 是要删除 chunk 的序号。
markNAND 在 yaffs 中使用 yaffs2 中不使用。
lyn 设置为当前行号用于调试。
*/
{
    int block;
    int page;
```

```
    yaffs_ExtendedTags tags;
    yaffs_BlockInfo *bi;

    if (chunkId <= 0)
        return;

    dev->nDeletions++;
    block = chunkId / dev->nChunksPerBlock;     //要删除 chunk 指向哪个 block
    page = chunkId % dev->nChunksPerBlock;
                                  //要删除 chunk 具体指向 block 中的哪个 page


    if (!yaffs_CheckChunkBit(dev, block, page))
                                  //检查该 block 和 page 是否为可删除
        T(YAFFS_TRACE_VERIFY,
            (TSTR("Deleting invalid chunk %d"TENDSTR),
             chunkId));

    bi = yaffs_GetBlockInfo(dev, block);        //获得 block 信息

    T(YAFFS_TRACE_DELETION,
      (TSTR("line %d delete of chunk %d" TENDSTR), lyn, chunkId));

    if (markNAND &&
       bi->blockState != YAFFS_BLOCK_STATE_COLLECTING && !dev->isYaffs2) {
        /*函数 yaffs_InitialiseTags()通过调用 memset 来设置 tags 为 0,同时将 tags
中的 validMarker0 设置为 0xAAAAAAAA, validMarker1 设置为 0x55555555;*/
        yaffs_InitialiseTags(&tags);
        tags.chunkDeleted = 1;                  //标记为被删除
        yaffs_WriteChunkWithTagsToNAND(dev, chunkId, NULL, &tags);
                                                //将标记 tags 写入块
        yaffs_HandleUpdateChunk(dev, chunkId, &tags);      //更新处理
    } else {
        dev->nUnmarkedDeletions++;              //如果是 yaffs2 只增加计数
    }
    if (bi->blockState == YAFFS_BLOCK_STATE_ALLOCATING ||
                                                //状态为正在分配
        bi->blockState == YAFFS_BLOCK_STATE_FULL ||        //状态为满
        bi->blockState == YAFFS_BLOCK_STATE_NEEDS_SCANNING ||
                                                //扫描 block 真正的状态
        bi->blockState == YAFFS_BLOCK_STATE_COLLECTING) {  //状态为正在回收
        dev->nFreeChunks++;                                //空闲块计数加 1
        yaffs_ClearChunkBit(dev, block, page);             //清楚块标记设置
        bi->pagesInUse--;                                  //使用页计数减 1
        if (bi->pagesInUse == 0 &&
            !bi->hasShrinkHeader &&
            bi->blockState != YAFFS_BLOCK_STATE_ALLOCATING &&
            bi->blockState != YAFFS_BLOCK_STATE_NEEDS_SCANNING) {
             yaffs_BlockBecameDirty(dev, block);           //标记为脏数据
        }
    }
}
```

### 4．YAFFS与JFFS的比较

YAFFS 和 JFFS 都提供了写均衡，垃圾收集等操作。同时在稳定性、垃圾收集速度、

储存容量等特性方面具有以下区别：

- JFFS 是一种日志文件系统，采用日志机制保证文件系统的稳定性。YAFFS 仅仅借鉴了日志系统的思想，不提供日志机制，所以稳定性不如 JFFS，但是资源占用少。
- JFFS 中使用多级链表管理需要回收的脏块，采用系统生成伪随机变量计算要回收的块，这种方法能提高硬件的写均衡，在 YAFFS 中是从头到尾对块进行扫描，所以在垃圾收集上 JFFS 的速度较慢，但是能延长 NAND 器件的寿命。
- JFFS 支持文件压缩，适合存储容量较小的系统；YAFFS 不支持压缩，更适合存储容量大的系统。

### 5.2.3　Cramfs 文件系统（Compressed ROM File System）

Cramfs 是一个压缩式的文件系统，不必一次性将文件系统中的全部内容解压缩到内存中，而只是在系统需要访问某个位置的数据时，先计算出该数据压缩后在 Cramfs 中所存的位置，再将该数据即时解压缩到 RAM 中，最后通过访问内存来读取文件系统中需要的数据。Cramfs 中的解压缩及解压缩之后的内存中数据存放位置，都是由 Cramfs 文件系统本身进行维护的，用户不需要了解具体的实现细节，因此这种方式增强了透明度，既方便，又节省了存储空间。

#### 1．Cramfs文件系统的特点

在 Cramfs 文件系统中，每一页（4KB）被单独压缩，可以随机页访问，其压缩比高达 2：1，节省了嵌入式系统 Flash 存储空间，系统使用低容量的 Flash 存储相同的文件，因此降低了系统的成本。另外，它的速度快，效率高，其只读特性有利于保护文件系统遭受破坏，提高了系统的可靠性。

Cramfs 的特性如下：

- 系统访问数据时采用实时解压缩方式，其解压缩算法复杂，因此解压缩过程有延迟。
- Cramfs 的数据都是经过处理、打包的，对数据进行写操作比较困难。所以 Cramfs 不支持写操作，这一特性适合嵌入式应用中使用 Flash 存储文件系统的场合。
- 在 Cramfs 中文件最大不能超过 16MB。
- 支持组标识（gid）。mkcramfs 处理掉 gid 的高 8 位，保留 gid 的低 8 位，因此只有 gid 的低 8 位是有效的。
- 支持硬链接，但是 Cramfs 不能处理多条链接，硬链接的文件属性中，链接数始终为 1。
- Cramfs 的目录中，没有“.”和“..”这两项。因此，Cramfs 中的目录链接数通常也仅有一个。
- Cramfs 中不保存文件的时间戳（timestamps）信息。正在使用的文件由 inode 保存在内存中，其时间可以暂时变更为最新时间，但是不会保存到 Cramfs 文件系统中。
- 当前版本的 Cramfs 只支持 PAGE_CACHE_SIZE 为 4096 的内核。因此，如果发现 Cramfs 不能正常读写的时侯，可以修改 mkcramfs.c 中的宏定义。

注意：对于上面特性的描述，具体细节可以查看 Cramfs.txt 文档中的 Usage Notes 描述。

2．Cramfs文件系统的优点和缺点

Cramfs 文件系统的优点有：压缩比较高，占用内存空间少；其缺点就是只能进行读操作，不支持写操作。

### 5.2.4　Romfs 文件系统（ROM File System）

Romfs 是一种简单的、紧凑的、只读的文件系统，不支持动态擦写保存功能，采用顺序存储方式，所有的数据包括目录、链接等都按照目录树的顺序进行存放。与 EXT2 等较大型的文件系统而言，Romfs 非常节省空间。通常 Romfs 用在嵌入式设备中作为根文件系统，或者用于保存 boot loader 以便引导系统启动。

因为 Romfs 是一种只读的文件系统，使用顺序存储方式，所有数据都是顺序存放的。它的数据存储方式决定了无法对 Romfs 进行写操作。因此 Romfs 中的数据一旦确定就无法修改，Romfs 只能作为一种只读文件系统。由于采取了顺序存放策略，Romfs 中每个文件的数据都能连续存放，读取过程中只需要一次寻址操作，就可以对整块数据进行读取，因此 Romfs 中读取数据效率很高。

Romfs 有两个结构，代码比较简单，在 romfs_fs.h 中定义如下：

```
struct romfs_super_block {
    __be32 word0;
    __be32 word1;
    __be32 size;
    __be32 checksum;
    char name[0];       /* volume name */
};
```

结构体 romfs_super_block 用于识别 Romfs 文件系统，大小为 512 字节，字段 word0 的初始值为‘-’，‘r’，‘o’，‘m’，字段 word1 的初始值为‘-’，‘1’，‘f’，‘s’，通过这两个字段系统可以确定这是一个 Romfs 文件系统。字段 size 记录整个文件系统的大小，理论上 Romfs 大小最多可以达到 4GB。checksum 字段是前 512 字节的校验和，用于确认整个文件系统结构数据的正确性。前 4 个字段占 16 字节，剩下的都可以用作文件系统的卷名，如果整个首部不足 512 字节部分采用 0 填充，保证首部遵循 16 字节对齐原则。

```
struct romfs_inode {
    __be32 next;        /* low 4 bits see ROMFH_ */
    __be32 spec;
    __be32 size;
    __be32 checksum;
    char name[0];
};
```

结构体 romfs_inode 是 Romfs 的文件结构。 next 字段指定下一个文件的偏移地址，该地址的后 4 位是保留的，用于记录文件模式信息，其中前两位标识文件类型，后两位标识该文件是否为可执行文件。因此 Romfs 用于文件寻址的 bit 数实际上只有 28bit，所以 Romfs 中文件大小不能超过 256MB（$2^{28}$）。spec 字段标识该文件类型，目前 Romfs 支持的文件

类型包括普通文件、目录文件、符号链接、块设备和字符设备文件。size 字段指明文件大小；checksum 字段是文件名和填充字段的校验和；name 字段是文件名首地址，文件名长度必须保证遵循 16 字节对齐原则，不足部分可用 0 填充。

对于 Romfs 文件系统的注册过程，读者可以查看 fs/romfs/inode.c 文件。其注册过程类似简单设备驱动注册过程。

## 5.3　基于 RAM 的文件系统

基于 RAM 文件系统的优点就是读写速度非常快，而缺点就是当系统复位后会丢失所有数据。下面分别简单介绍基于 RAM 的文件系统特点。

### 1．Ramdisk

Ramdisk 是划分一块固定大小的内存作分区来使用，它不是一个实际的文件系统，而是将实际的文件系统装入内存的一种策略，并且可以作为根文件系统。将一些经常被访问而又不会更改的文件（如只读的根文件系统）通过 Ramdisk 放在内存中，可以明显地提高系统的性能。

在 Linux 的启动阶段，initrd 提供了一套机制，将内核映像和根文件系统一起加载到内存中。在 initrd 机制中还会指定文件系统的起始地址、大小等参数，这些参数会通过 Bootloader 传递给内核。

### 2．Ramfs/Tmpfs

Ramfs 是 Linus Torvalds 开发的一种基于内存的文件系统，工作于虚拟文件系统（VFS）层，不能进行格式化，可以创建多个，在创建时可以指定其最大使用的内存大小（VFS 可看成是一种内存文件系统，统一了文件在内核中的表示方式，并对磁盘文件系统进行了缓冲）。

Ramfs/Tmpfs 文件系统把所有的文件都放在 RAM 中，所以读/写操作发生在 RAM 中，可以用 Ramfs/Tmpfs 来存储一些临时性或经常要修改的数据，例如/tmp 和/var 目录，这样既避免了对 Flash 存储器的读写损耗，也提高了数据读写速度。

Ramfs/Tmpfs 相对于传统的 Ramdisk 的不同之处主要在于其不能被格式化，文件系统大小可随所含文件内容大小变化。

## 5.4　文件系统的制作

5.3 节介绍了常用的文件系统的特点，以及如何根据硬件方案选择合适的文件系统。本节将介绍如何制作选择的文件系统。另外，Busybox 集合了很多工具，编译起来也非常方便。在讲解制作文件系统的时候，也会介绍 busybox 的编译和安装过程，介绍制作文件系统时，会详细介绍 Ramdisk 和 YAFFS 2 文件系统的制作。

## 5.4.1　制作 Ramdisk 文件系统

制作根文件系统需要有如下目录：/dev、/bin、/usr、/sbin、/lib、/etc、/proc 和/sys。下面分别简单介绍各个目录中存放的文件。

（1）/dev 目录下存放的是设备文件，用于访问系统资源或设备，如串口、U 盘、硬盘、系统内存等。在 Linux 中所有的设备都被抽象成了文件，用户可以访问设备就像访问普通文件一样。在/dev 目录下，每个文件可用 mknod 建立。/dev 目录下主要的设备文件包括以下几个。

- /dev/console：系统控制台设备文件。
- /dev/hd IDE：接口硬盘设备文件。
- /dev/fd：软驱设备文件。
- /dev/sd：SCSI 接口磁盘驱动器文件。
- /dev/tty：设备虚拟控制台。
- /dev/ttyS*：串口设备文件。

（2）/bin、/usr/bin、/usr/sbin、/sbin 存放的是二进制可执行文件，这部分内容通常通过编译 busybox 获得。

（3）/lib 用于存放动态链接库。

（4）/etc 是用来存放初始化脚本和其他配置文件的。启动脚本位于/etc/rc.d/init.d 中，系统最先运行的服务是那些放在/etc/rc.d 目录下的文件，运行级别在文件/etc/inittab 中指定。

（5）/proc 是用来挂载存放系统信息虚拟文件的系统，不保存在系统硬盘中，是内存的映射。它包含一些和系统相关的信息，如 CPU 的信息。

（6）/sys 该目录下安装了 2.6 内核中新出现的 sysfs 文件系统，sysfs 集成了 3 种文件系统的信息：针对进程信息的 proc 文件系统、针对设备的 devfs 文件系统及针对伪终端的 devpts 文件系统。sysfs 是内核设备树的一个直观反映。当一个内核对象被创建时，会在内核对象子系统中创建对应的文件和目录。

下面将详细介绍 Ramdisk 的制作过程。

### 1．建立根文件目录

前面提到过根文件目录主要包括/dev、/bin、/usr、/sbin、/lib、/etc、/proc、/sys、/var 和/tmp。下面给出建立根文件目录的命令：

```
#cd /usr/local
#mkdir rootfs
#cd rootfs
#mkdir bin dev etc lib proc sbin tmp usr var sys
#chmod 777 tmp
#mkdir usr/bin usr/lib usr/sbin
#mkdir var/lib var/lock var/log var/run var/tmp
#chmod 777 var/tmp
```

### 2．编译Busybox

编译 Busybox 可以得到绝大多数目录和工具，可以简化设计和开发时间。在下载和使

用 busybox 时，注意要使用稳定版本（stable）。例如，Busybox 1.16.1 是稳定版本，而 Busybox 1.16.0 是非稳定版本，建议读者在初学时使用稳定版本。

编译 Busybox 前先必须对需要的工具进行配置，通过图形界面选择工具，选择的原则是尽量选择必要的工具。下面是解压和进入配置界面命令：

```
#tar jxvf busybox-1.16.1.tar.bz2
#cd busybox-1.16.1
#make menuconfig
```

（1）进入配置界面后，选择 Busybox Settings-->Build Options--->，在该窗口中设置将 Busybox 编译成静态库，选择交叉编译器，如图 5.2 所示。

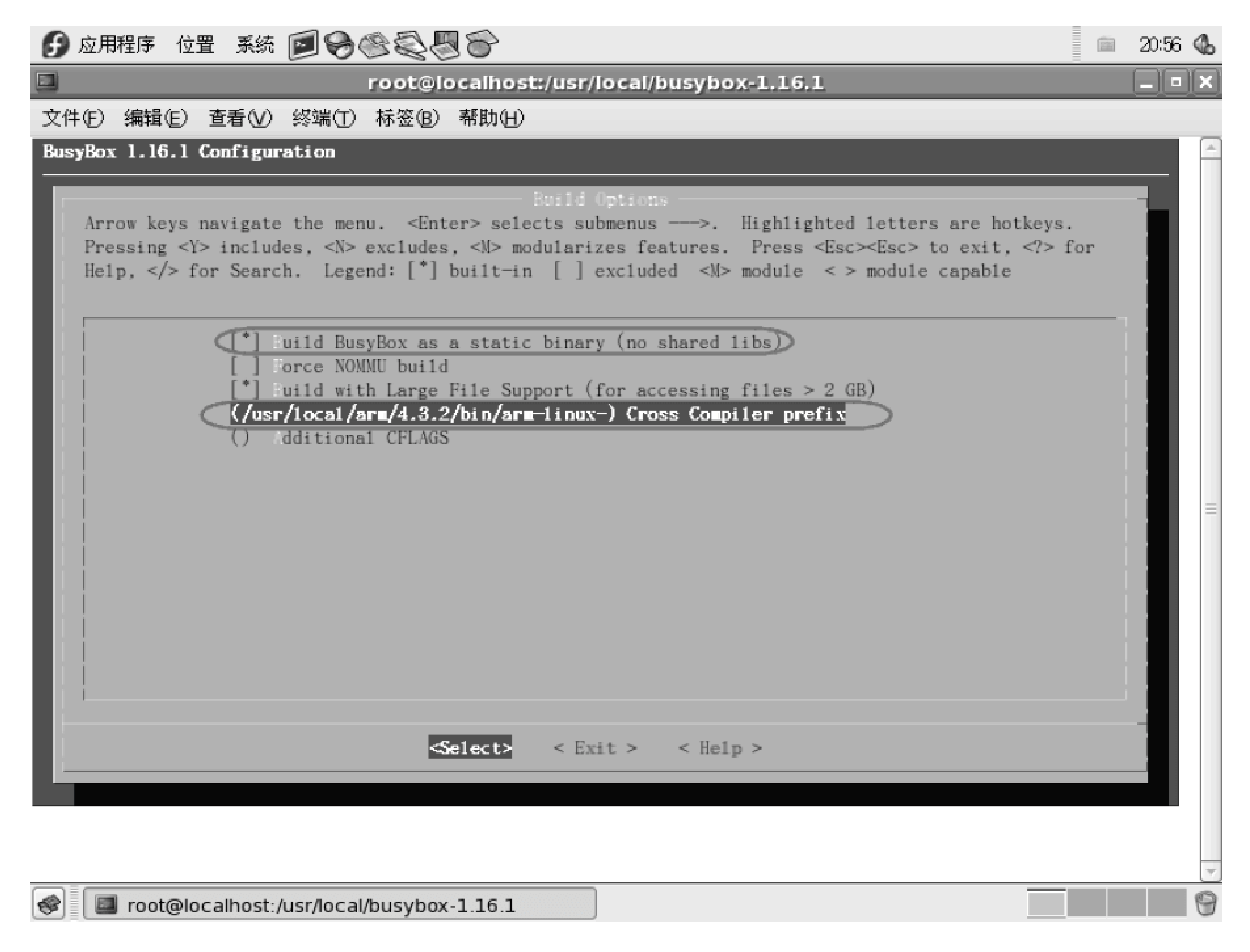



图 5.2　设置编译选项 Build Options

（2）配置安装选项，选择 Busybox Settings-->Installation Options--->，进入 Installation Options 窗口后设置 busybox 的安装目录为/usr/local/rootfs，即前面创建的根文件目录，如图 5.3 所示。

（3）配置关于档案工具选项（Archival Utilities），该窗口中有常用的压缩（bzip2）、解压（bunzip2）和安装软件包工具（rpm）等。可以选择常用的工具，也可以按照默认的选择进行配置，如图 5.4 所示。

（4）配置核心工具选项（Coreutils），该窗口中包括打印日历（cal），修改权限（chmod），复制（cp），移动文件（mv）等，可以选择常用的工具，也可以按照默认的选择进行配置，如图 5.5 所示。

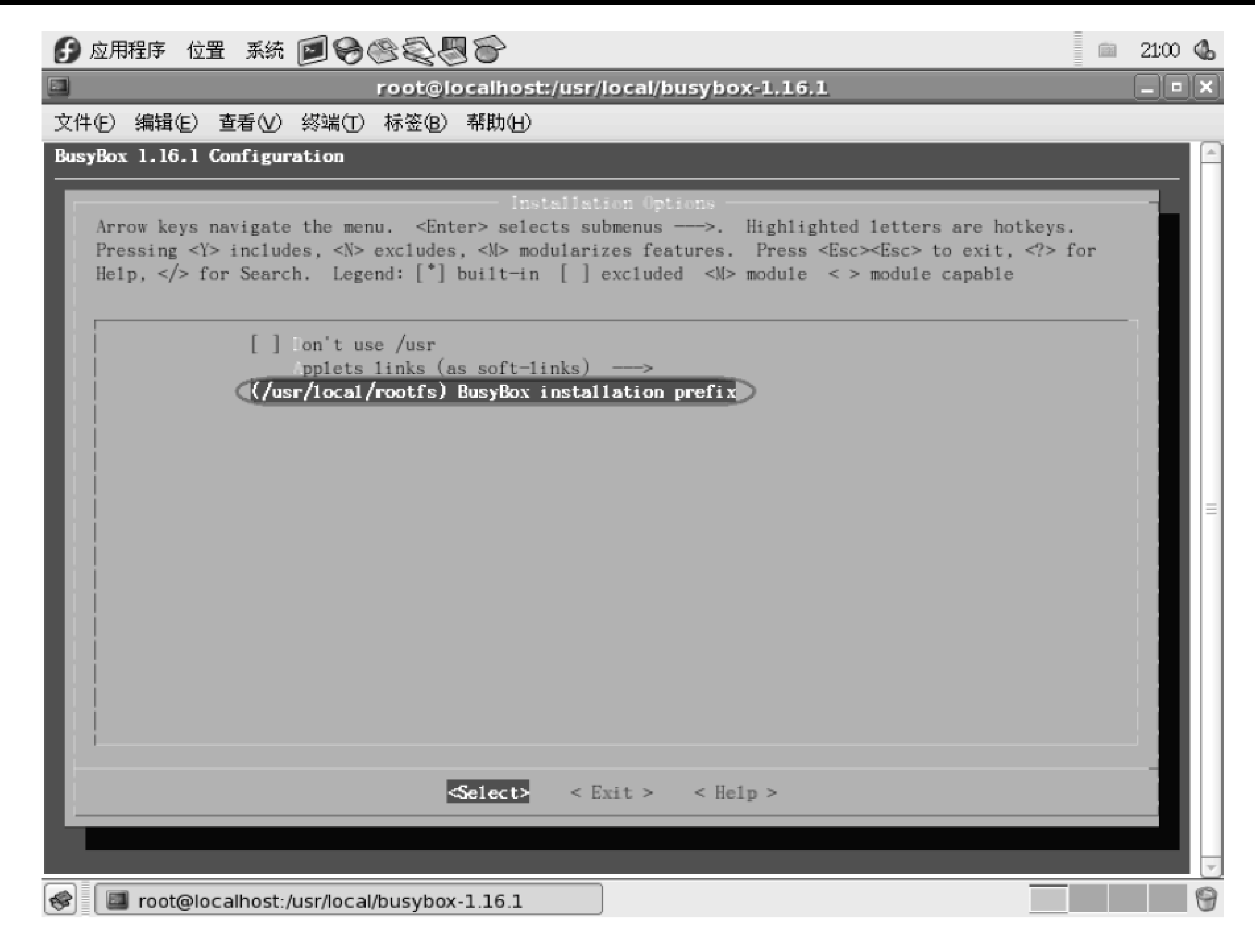



图 5.3　设置安装选项（Installation Options）

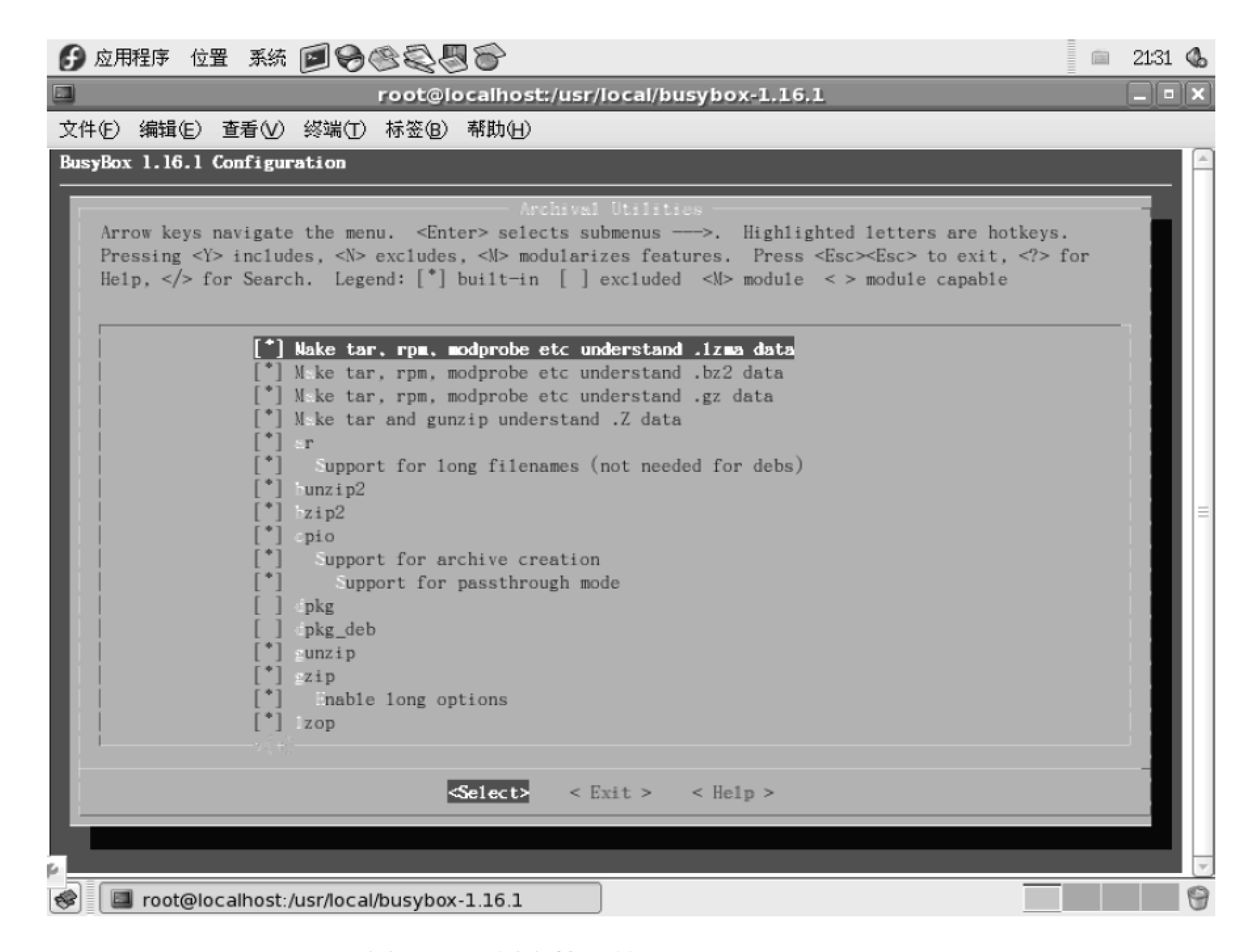



图 5.4　配置文档工具（Archival Utilities）

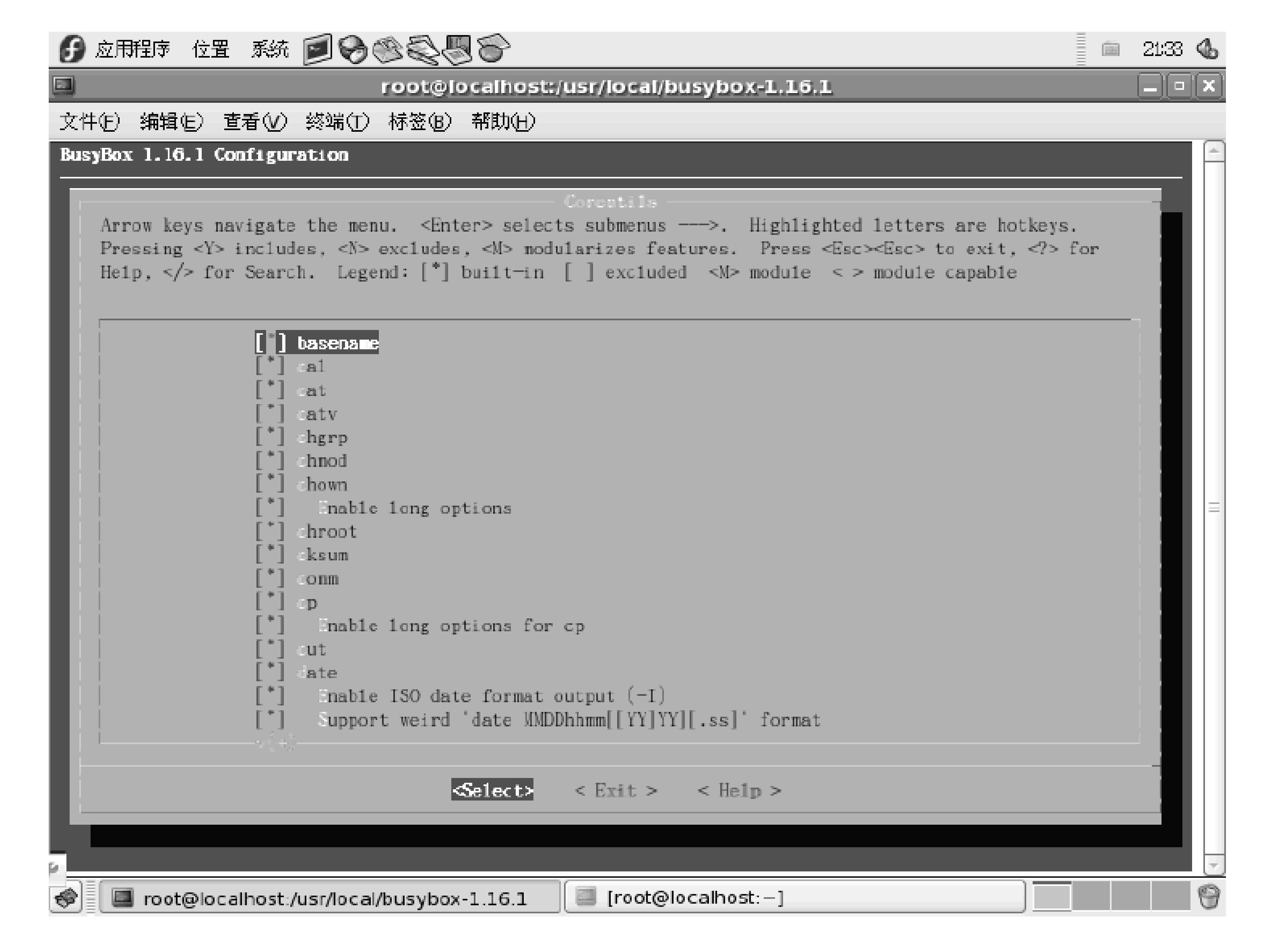



图 5.5　配置核心工具选项（Coreutils）

（5）配置控制台工具（Console Utilities），该窗口中的工具在实际中用的比较少，常用的有清除控制台（clear）、重置（reset）控制台等，读者可以根据需要选择，如图 5.6 所示。




图 5.6　配置控制台工具（Console Utilities）

（6）Debian Utilities 和 Mail Utilities，这两项工具在嵌入式系统中基本没有用到，读者可以不用配置这两个选项。

（7）配置 Editors 时，可以只选 VI 和 diff 工具。

（8）必须配置初始化工具（Init Utilities），并且在该窗口中一定要选择 Support reading an inittab file，支持 init 读取/etc/inittab 配置文件，如图 5.7 所示。

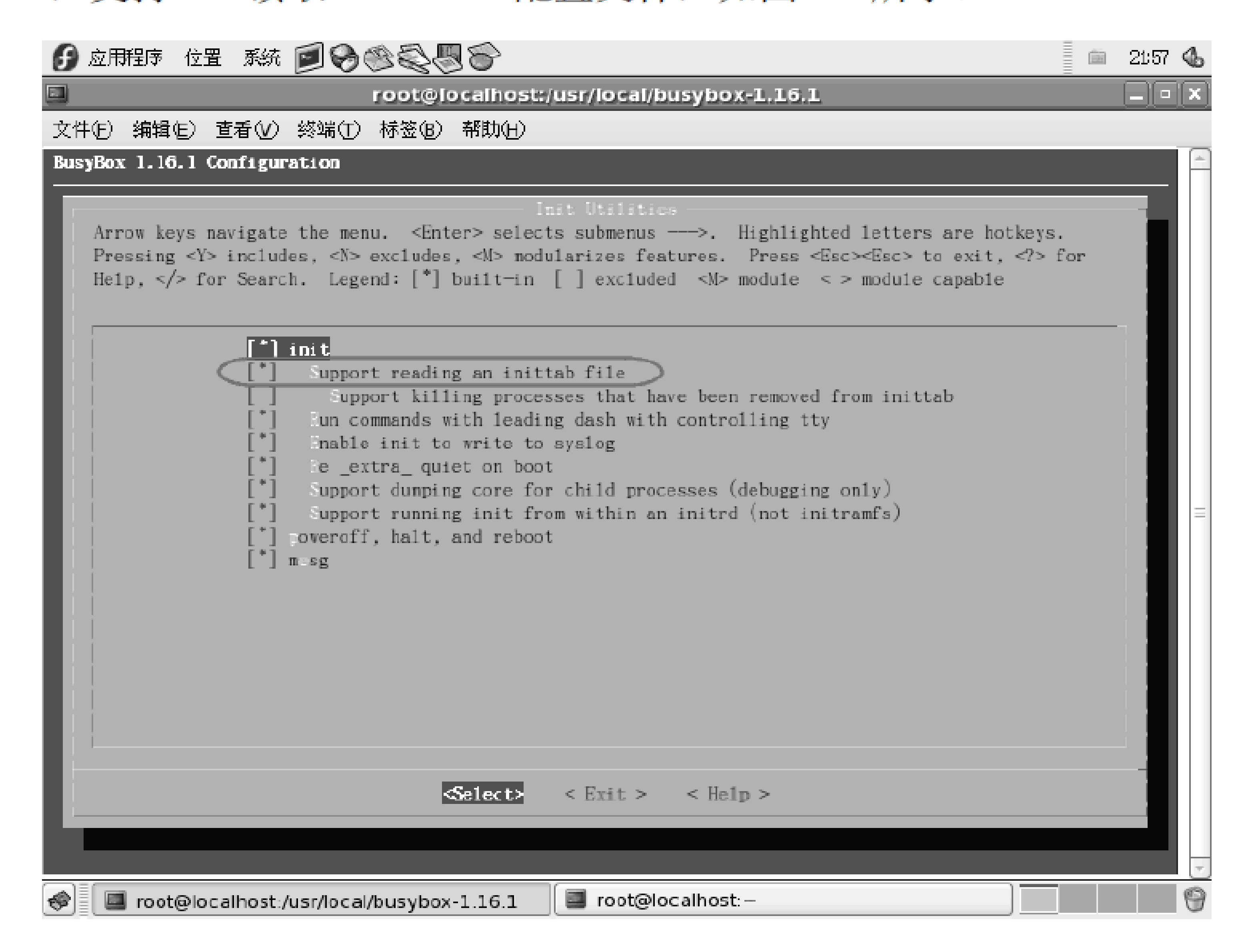



图 5.7　配置初始化工具（Init Utilities）

（9）必须配置网络工具（Networking Utilities），要与开发板进行通信，或者上传文件到开发板上时，需要通过网络进行传输。因此，需要有设置 IP 工具（ifconfig），文件传输工具（FTP）等，可以不用支持 IPv6、ARP 等工具，如图 5.8 所示。

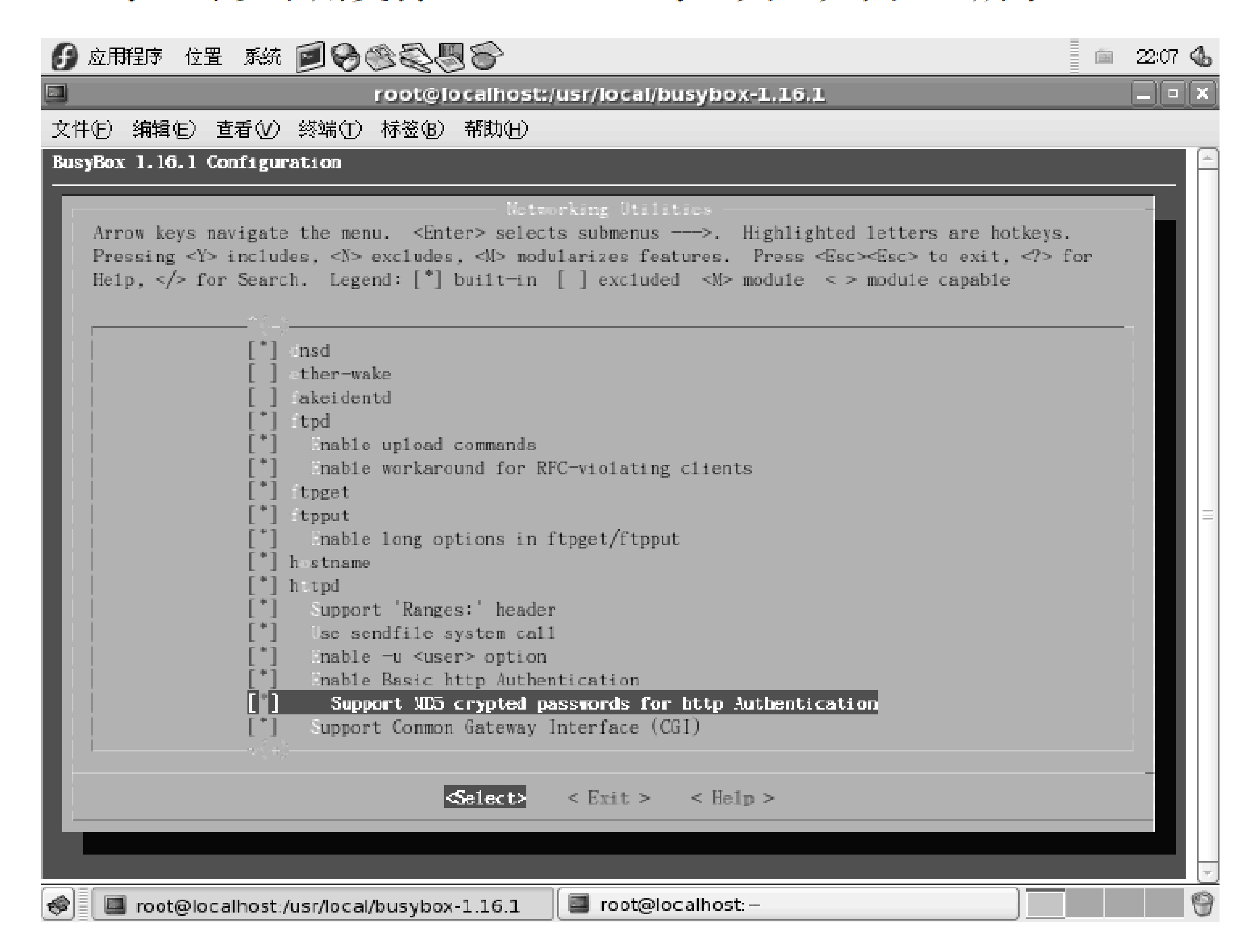



图 5.8　配置网络工具（Networking Utilities）

（10）必须配置 Shell 工具，选择命令 Choose your default shell (ash)进入 Choose your

default shell 窗口，选择 ash，如图 5.9 所示。

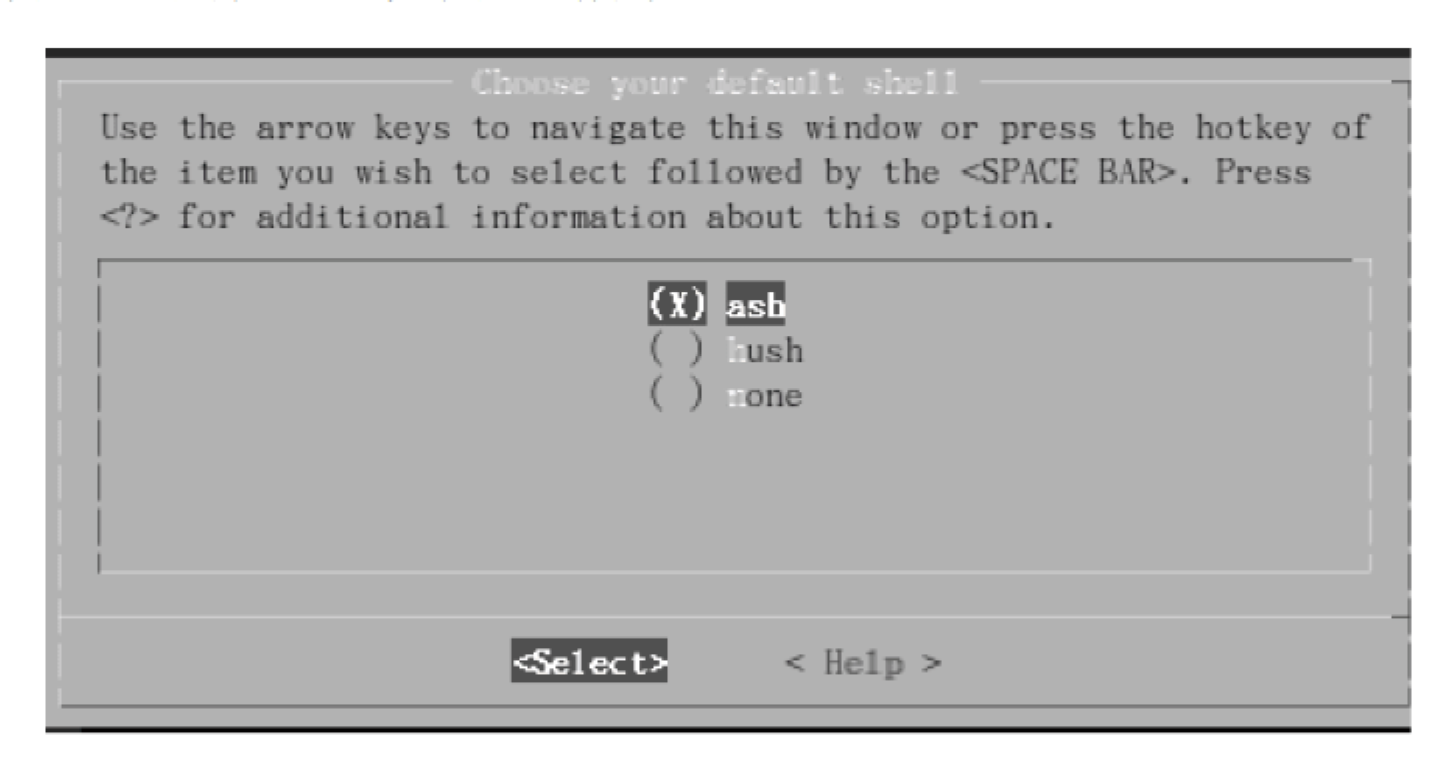



图 5.9　配置 Shell 工具

（11）保存配置，选择 Save Configuration to an Alternate File，退出配置窗口后执行下面的命令进行编译安装 busybox 到/usr/local/rootfs 目录下。

```
#make ARCH=arm CROSS_COMPILE=arm-linux- install
```

### 3．将交叉编译器库复制到 rootfs/lib下

（1）将交叉编译器目录下库文件复制到 rootfs/lib 中时，注意查看所复制的目录下是否有 libm、libpthread 等常用库。进入/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib 下，查看目录下的库文件，是否存在需要的库文件。

```
#cd /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib
#ls
ld-2.8.so                libcrypt-2.8.so          libm.so.6
                         libnss_hesiod-2.8.so     libresolv-2.8.so
ld-linux.so.3            libcrypt.so.1            libnsl-2.8.so
                         libnss_hesiod.so.2       libresolv.so.2
libanl-2.8.so            libc.so.6                libnsl.so.1
                         libnss_nis-2.8.so        librt-2.8.so
libanl.so.1              libdl-2.8.so             libnss_compat-2.8.so
                         libnss_nisplus-2.8.so    librt.so.1
libBrokenLocale-2.8.so   libdl.so.2               libnss_compat.so.2
                         libnss_nisplus.so.2      libSegFault.so
libBrokenLocale.so.1     libgcc_s.so              libnss_dns-2.8.so
                         libnss_nis.so.2          libthread_db-1.0.so
libc-2.8.so              libgcc_s.so.1            libnss_dns.so.2
                         libpcprofile.so          libthread_db.so.1
libcidn-2.8.so           libm-2.8.so              libnss_files-2.8.so
                         libpthread-2.8.so        libutil-2.8.so
libcidn.so.1             libmemusage.so           libnss_files.so.2
                         libpthread.so.0          libutil.so.1
```

（2）执行库文件的复制过程。复制完成后进入/usr/local/rootfs/lib 查看是否复制了需要的库文件。

```
#cd  /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib
#for file in libc libcrypt libdl libm libpthread libresolv libutil
>do
```

```
>cp $file-*.so /usr/local/rootfs/lib
>cp -d $file.so.[*0-9] /usr/local/rootfs/lib
>done
#cp -d ld*.so* /usr/local/rootfs/lib
# cd /usr/local/rootfs/lib
#ls
ld-2.8.so    libcrypt-2.8.so   libdl-2.8.so   libm.so.6    libresolv-2.8.so
libutil.so.1
ld-linux.so.3 libcrypt.so.1  libdl.so.2  libpthread-2.8.so libresolv.so.2
libc-2.8.so   libc.so.6     libm-2.8.so  libpthread.so.0  libutil-2.8.so
```

### 4．建立所需设备文件

需要的设备文件结点包括控制台 console、内存 mem 等。建立各个设备结点的参数包括设备类型、主设备号和次设备号。建立结点命令如下：

```
# cd /usr/local/rootfs/dev
# mknod console c 5 1
# mknod full c 1 7
# mknod kmem c 1 2
# mknod mem c 1 1
# mknod null c 1 3
# mknod port c 1 4
# mknod random c 1 8
# mknod urandom c 1 9
# mknod zero c 1 5
# for i in `seq 0 7`; do mknod loop$i b 7 $i; done
# for i in `seq 0 9`; do mknod ram$i b 1 $i; done
# ln -s ram1 ram
# mknod tty c 5 0
# for i in `seq 0 9`; do mknod tty$i c 4 $i; done
# for i in `seq 0 9`; do mknod vcs$i b 7 $i; done
# ln -s vcs0 vcs
# for i in `seq 0 9`; do mknod vcsa$i b 7 $i; done
# ln -s vcsa0 vcsa
```

注意：符号``并非键盘上的单引号，而是键盘左上方的波浪号对应的键。建立完成后可以查看在/usr/local/rootfs/dev 目录下建立的设备结点有：

```
console  loop1  loop5  null  ram1  ram5  ram9   tty1  tty5  tty9    vcs1
vcs5  vcs9   vcsa2  vcsa6  zero
full     loop2  loop6  port  ram2  ram6  random tty2  tty6  urandom vcs2
vcs6  vcsa   vcsa3  vcsa7
kmem     loop3  loop7  ram   ram3  ram7  tty    tty3  tty7  vcs     vcs3
vcs7  vcsa0  vcsa4  vcsa8
loop0    loop4  mem    ram0  ram4  ram8  tty0   tty4  tty8  vcs0    vcs4
vcs8  vcsa1  vcsa5  vcsa9
```

### 5．建立文件系统映像文件

准备目标系统启动所需要的文件 rcS、inittab 和 fstab。这 3 个文件是制作文件系统最重要的文件。下面给出各个文件的内容。

（1）/etc/init.d/rcS：挂载/etc/fstab 指定的文件系统。

```
#! /bin/sh
```

```
/bin/mount -a
```

（2）/etc/inittab：init 进程的配置文件。

```
::sysinit:/etc/init.d/rcS
::askfirst:-/bin/bash
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

（3）etc/fstab：指定需要挂载的文件系统。

```
proc /proc proc defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
sysfs /sys sysfs defaults 0 0
tmpfs /dev tmpfs defaults 0 0
var /dev tmpfs defaults 0 0
```

### 6．建立文件系统映像文件

建立根文件系统挂载点

```
# mkdir /mnt/ramdisk
```

建立大小为 8192 的根文件系统

```
# mke2fs -vm0 /dev/ram 8192
细节中打印的细节信息中包括块的个数，块的大小，结点个数等信息。
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
2048 inodes, 2048 blocks
0 blocks (0.00%) reserved for the super user
First data block=0
1 block group
32768 blocks per group, 32768 fragments per group
2048 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 30 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

挂载根文件系统

```
# mount -t ext2 /dev/ram  /mnt/ramdisk
```

对文件系统进行操作，将制作的文件系统拷贝到挂载点

```
# cp -af /usr/local/rootfs/* /mnt/ramdisk
```

退出/mnt/ramdisk 目录才能进行卸载

```
# cd /
```

卸载文件

```
# umount /mnt/ramdisk
```

文件系统生成

```
# dd if=/dev/ram of=ramdisk bs=1k count=8192
```

制作文件系统映像

```
# gzip -v9 ramdisk
```

生成的映像文件为 ramdisk，压缩后为 ramdisk.gz。

### 7．内核中支持RAM文件系统的初始化

在编译内核时，在 General setup 窗口中选择[*] Initial RAM filesystem and RAM disk (initramfs/initrd) support 如图 5.10 所示，同时在 Initramfs source 中传递初始化参数：

```
initrd=0x21100000,8000000 root=/dev/ram rw init=linuxrc console=ttyS0,
115200, mem=32M
```




图 5.10　配置 RAM 文件系统的初始化

## 5.4.2　制作 YAFFS2 文件系统

如果开发板只有 Nand Flash，那么选择最合适的文件系统为 YAFFS 文件系统。mini2440 只有 Nand Flash 没有 Nor Flash，因此选择的文件系统为 YAFFS2 文件系统。

### 1．制作文件系统时准备的源代码

内核源代码和交叉编译器读者可以根据自己的实际情况选择具体的对应版本，yaffs2.tar.gz 源码是必须的。

- linux-2.6.29.6.tar.bz2：内核源代码；
- yaffs2.tar.gz：YAFFS2 文件系统源代码；
- arm-linux-gcc-4.3.2.tgz：交叉编译工具；
- mkyaffs2image.tgz：制作 YAFFS2 文件系统工具。

### 2．解压源码

解压内核源码和交叉编译器的源码，并将 yaffs2.tar.gz 复制到内核源码的 fs 目录下进行解压。如果是第一次使用交叉编译器，那么应该在环境变量中添加交叉编译器的路径或者在/etc/profile 中添加交叉编译器路径并重新启动计算机。文件/etc/profile 中的交叉编译器的设置，例如：

```
# Path manipulation
if [ "$EUID" = "0" ]; then
    pathmunge /sbin
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
    pathmunge /usr/local/arm/4.3.2/bin
fi
```

### 3．修改内核顶层Makefile

在 Makefile 中设置目标平台为 arm，交叉编译器为 arm-linux。

```
# vi Makefile
ARCH ?= $(SUBARCH)
CROSS_COMPILE ?=
修改为
ARCH ?=arm
CROSS_COMPILE ?=arm-linux-
```

### 4．修改机器码

在 vivi 启动时如果机器码与设置的不一致会出现提示，在文件 arch/arm/tools/mach-types 中进行下面的修改。

```
# vi arch/arm/tools/mach-types
s3c2440 ARCH S3C2440 S3C2440 362
修改为
s3c2440 ARCH_S3C2440 S3C2440 782
```

### 5．修改时钟频率

修改 arch/arm/mach-s3c2440/mach-smdk2440.c 中的时钟为 12MHz，具体修改如下。

```
static void __init smdk2440_map_io(void)
{
    s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
    //s3c24xx_init_clocks(16934400);
    s3c24xx_init_clocks(12000000);//将频率设置为 12MHz
    s3c24xx_init_uarts(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
}
```

### 6．使内核支持YAFFS2

解压 yaffs2.tar.gz 后进入 YAFFS2 目录，在 YAFFS2 目录下有可执行文件 patch-ker.sh，执行如下命令：

```
# ./patch-ker.sh c /usr/local/arm/linux-2.6.29.6
```

执行该命令后，就会在 fs 的 Kconfig 和 Makefile 中增加对 YAFFS2 的编译选项的支持。在 fs/Kconfig 会自动添加：

```
# Patched by YAFFS
source "fs/yaffs2/Kconfig"
```

在 fs/Makefile 中会自动添加：

```
# Patched by YAFFS
obj-$(CONFIG_YAFFS_FS)          += yaffs2/
```

注意：这两部分内容也可以进行手动添加。添加的目的是在内核的文件系统选项中增加了对 YAFFS2 的支持选项。

回到内核的一级目录下运行 make menuconfig，对内核进行配置，配置中多了对 YAFFS2 文件系统支持的选项，选上该选项，如图 5.11 所示。

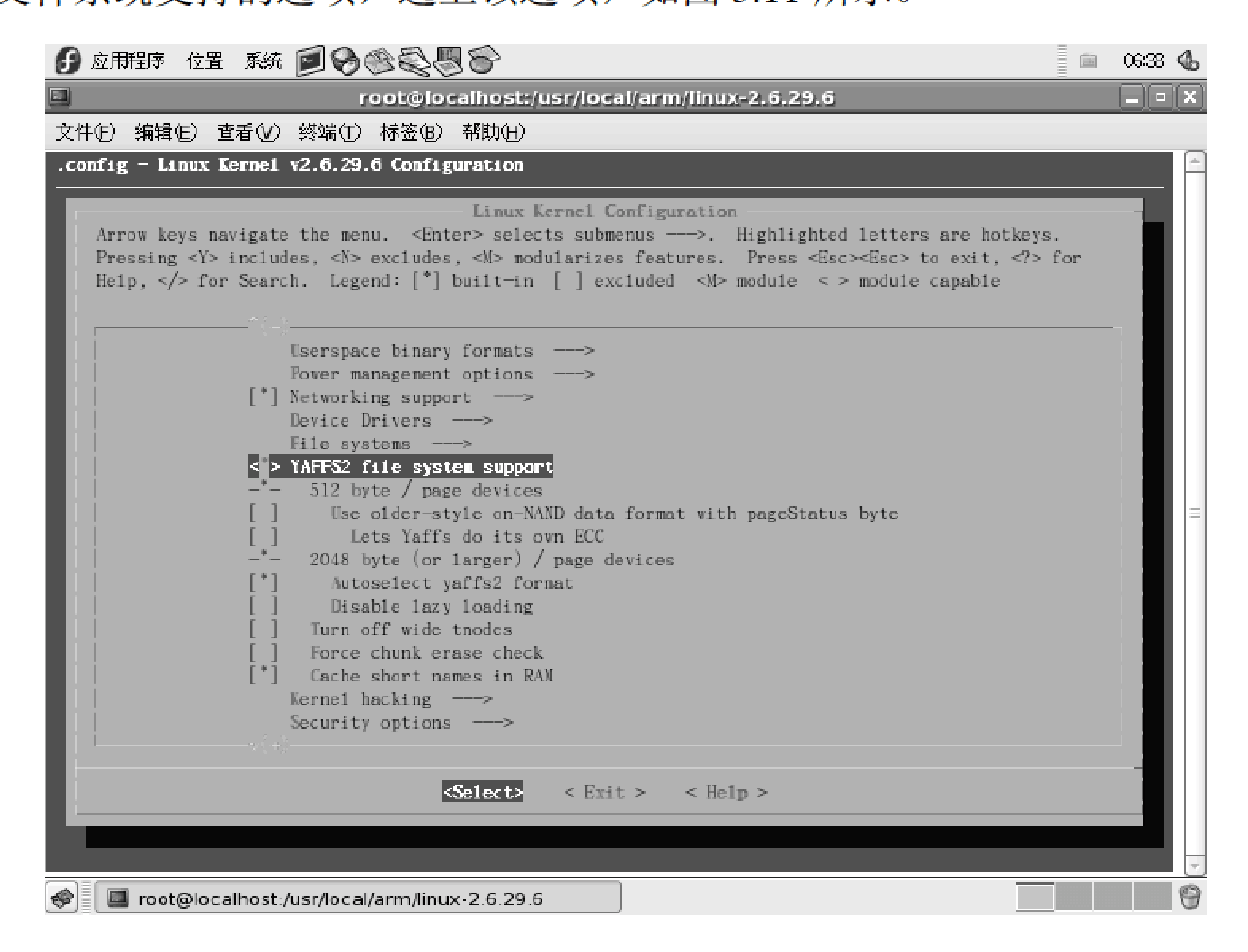



图 5.11　内核中增加对 YAFFS2 的支持

注意：执行 patch-ker.sh 时将语句 source "fs/yaffs2/Kconfig" 自动放在文件 fs/Kconfig 的末尾，即和文件系统并列的菜单。可以对其位置进行修改使之在文件系统菜单内。如将其加在 cramfs 选项的后面，在 fs/Kconfig 中代码如下，内核中对 YAFFS2 支持就出现在 cramfs 后面，如图 5.12 所示。

```
source "fs/cramfs/Kconfig"
source "fs/yaffs2/Kconfig"
```

进入内核配置界面后，选择命令 File systems　---> Miscellaneous filesystems　--->，进入 Miscellaneous filesystems 配置窗口，选择对 YAFFS2 文件系统的支持。

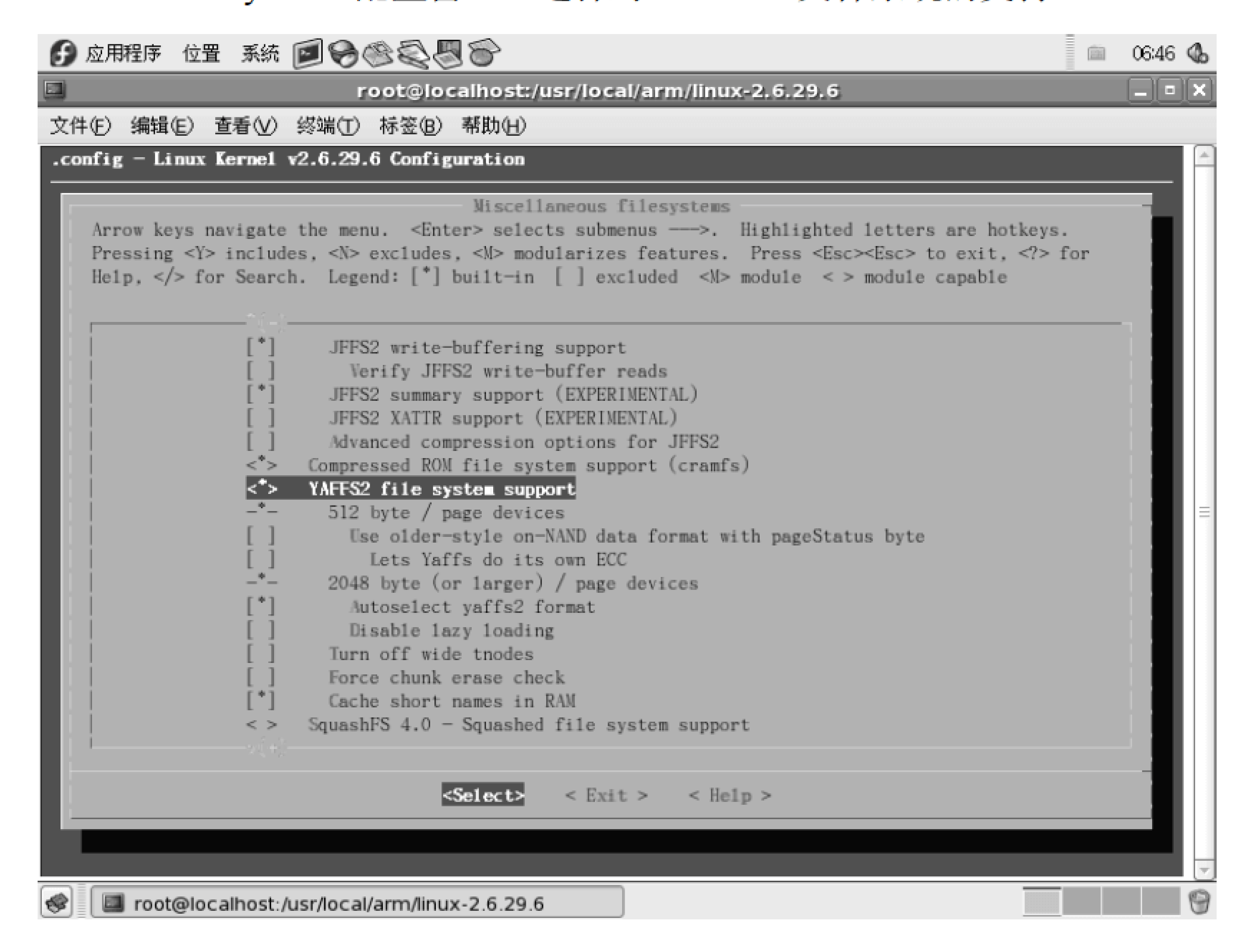



图 5.12　修改位置后的 YAFFS2 选项

### 7．使内核支持Mini2440

在内核的 System Type-->ARM system type 选项下，选择 Samsung S3C24xx 系列，如图 5.13 所示。如果读者的开发板不是 mini2440，那么就应该选择对应的处理器类型。然后在 S3C2440 Machines 中选择 Mini2440 支持选项。

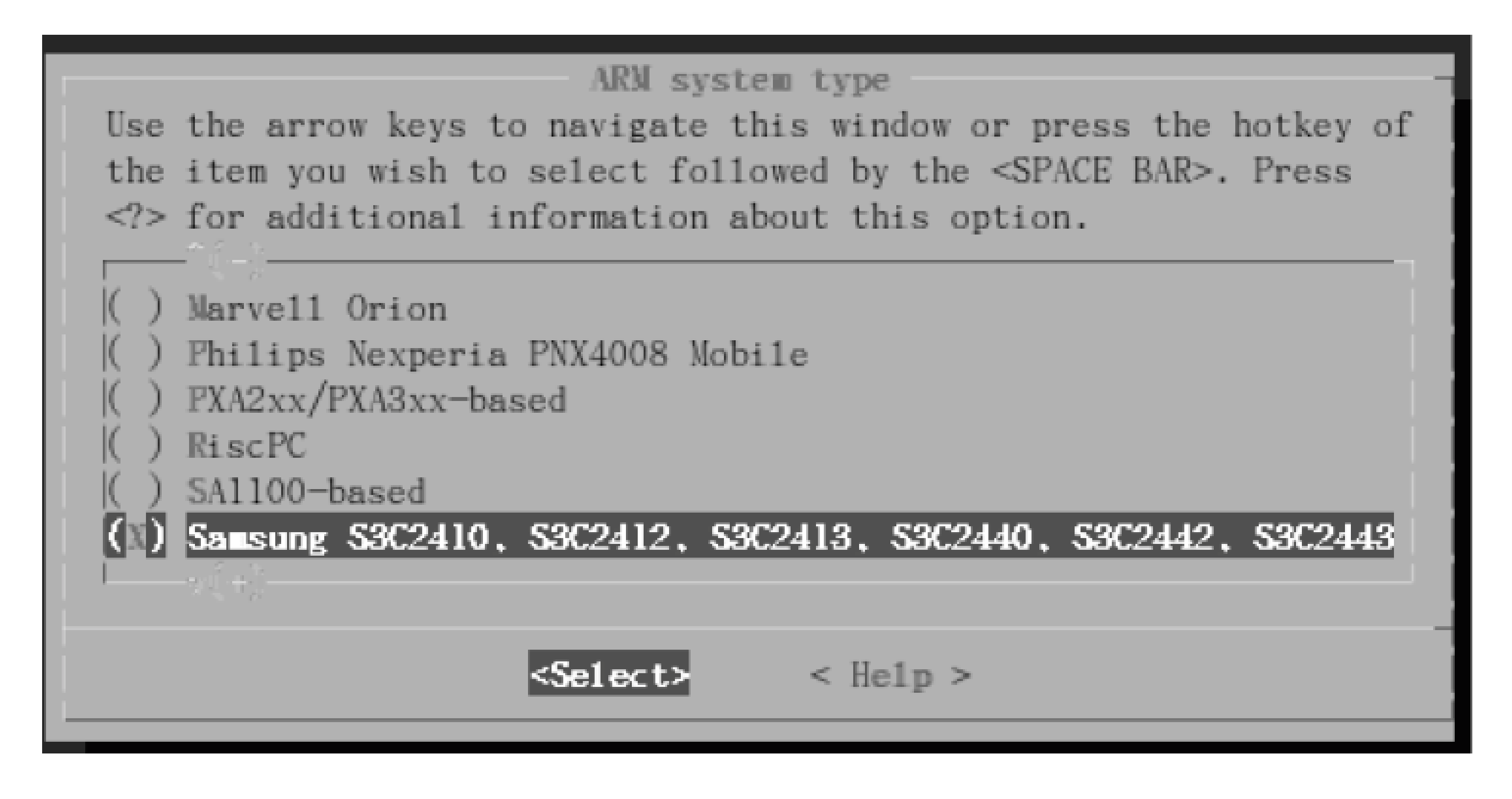



图 5.13　选择处理器类型

## 8．编译内核映像文件

执行 make zImage 生成内核的映像文件，如果遇到下面的错误可以执行 make distclean 进行清理，然后重新生成映像文件。

```
ERROR: the symlink include/asm points to asm-x86 but asm-arm was expected
      set ARCH or save .config and run 'make mrproper' to fix it
# make distclean
# make zImage
```

## 9．编译Busybox

编译 Busybox 的配置细节可以参考 5.2.1 节。这里可以将 Busybox 交叉编译安装在_install 文件中，如图 5.14 所示配置安装路径。

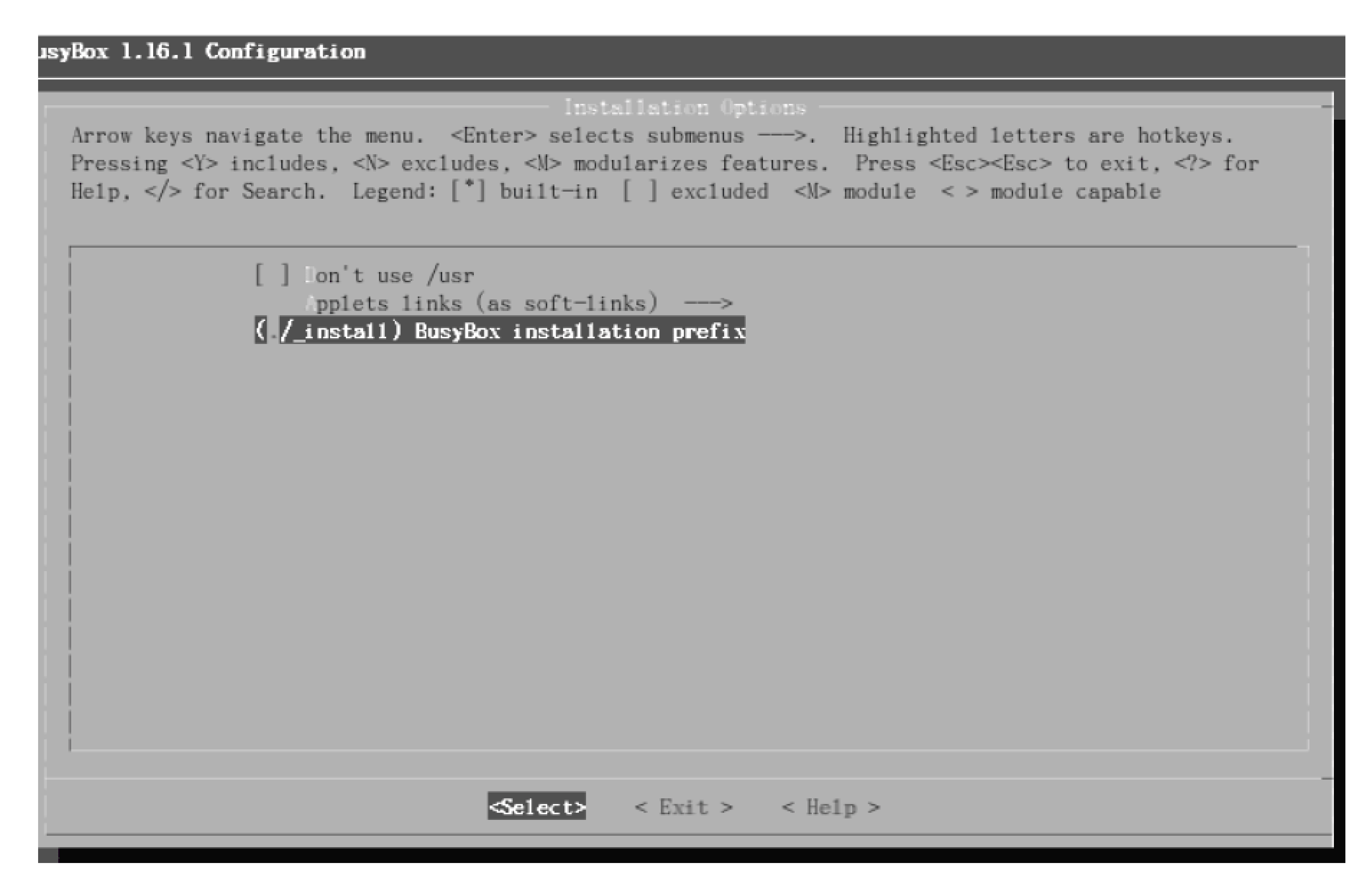



图 5.14　配置 Busybox 安装路径

在/usr/local 目录下新建 yaffs_root 文件夹，将 Busybox 的安装目录_install 中的文件 bin、linuxrc、sbin、usr 复制到 yaffs_root 目录下。

```
# mkdir /usr/local/yaffs_root
# cd /usr/local/busybox-1.16.1/_install
# cp -rf bin linuxrc sbin usr /usr/local/yaffs_root
```

## 10．为YAFFS文件系统准备lib库

将交叉编译器目录下的库文件全部复制到 lib 库目录下，–d 表示复制的时候包括链接文件一起复制过来。

```
#cp -d /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib/*so* ./lib
```

### 11．制作etc目录下必要的文件

etc 目录是文件系统中最重要的目录，系统配置的启动信息都在该目录下，下面分别给出必要的几个文件。

（1）/etc/inittab 文件。

```
::sysinit:/etc/init.d/rcS          #调用系统初始化文件
s3c2410_serial0::askfirst:-/bin/sh  #文件 drivers/serial/s3c2410.c 中指定了串口驱动名字 s3c2410_serial
::ctrlaltdel:/sbin/reboot          #重启
::shutdown:/bin/umount -a -r       #关机
```

该文件为 init 进程的配置文件。

（2）etc/init.d/rcS 文件。

```
#!/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
runlevel=S
prevlevel=N
umask 022
export PATH runlevel prevlevel
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
mkdir -p /var/lock
/bin/hostname -F /etc/sysconfig/HOSTNAME
```

该文件为可执行文件，完成后还要修改其权限为可执行。该文件功能包括指定环境变量、运行级别和挂载设备等。

（3）etc/profile 文件。

```
USER="`id -un`"
LOGNAME=$USER
PS1='[Yaffs_LiuG]# '
PATH=$PATH
HOSTNAME=`/bin/hostname`
export USER LOGNAME PS1 PATH
```

如果不配置 profile，移植完文件系统后，进入系统命令行头为空，效果如下：

```
# ls
bin         home        lost+found  proc        sys         var
dev         lib         mnt         root        tmp
etc         linuxrc     opt         sbin        usr
```

配置 profile 文件后，其效果如下：

```
[Yaffs@LiuG]# ls
bin         home        lost+found  proc        sys         var
dev         lib         mnt         root        tmp
etc         linuxrc     opt         sbin        usr
```

（4）etc/sysconfig/HOSTNAME 文件。

```
Yaffs@LiuG                  #指定 HOSTNAME,rcS 中调用该文件
```

（5）etc/fstab 文件。

```
proc /proc proc defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
sysfs /sys sysfs defaults 0 0
tmpfs /dev tmpfs defaults 0 0
var /dev tmpfs defaults 0 0
```

该文件指明需要挂载的文件系统。

#### 12．制作YAFFS映像文件

解压制作文件系统工具 mkyaffs2image.tgz，解压后该工具自动安装在目录 usr/sbin/下。使用 mkyaffs2image 将上面制作的文件系统制作成映像文件。

```
# tar zxvf mkyaffs2image.tgz
# mkyaffs2image yaffs_root yaffs_root.img
```

### 5.4.3　制作 JFFS2 文件系统

制作 JFFS2 文件系统是通过工具 mkfs.jffs2 将文件系统目录制成映像文件。制作工具 mkfs.jffs2 需要编译 zhb 库和 mtd-utils，下面详细介绍其制作过程。

#### 1．内核配置MTD驱动支持和JFFS2支持

从图 5.1 可以看出 YAFFS2 自带 MTD 驱动，而 JFFS2 文件系统则需要在内核中配置 MTD 驱动支持。内核也必须支持 JFFS2 文件系统。

在编译内核时选择 Device Drivers ---> Memory Technology Device (MTD) support --->，进入 Memory Technology Device 配置窗口，如图 5.15 所示。




图 5.15　配置 MTD 驱动支持

在编译内核时选择 File systems --->Miscellaneous filesystems--->，进入 Miscellaneous filesystems 配置窗口选择支持 JFFS2 文件系统，如图 5.16 所示。

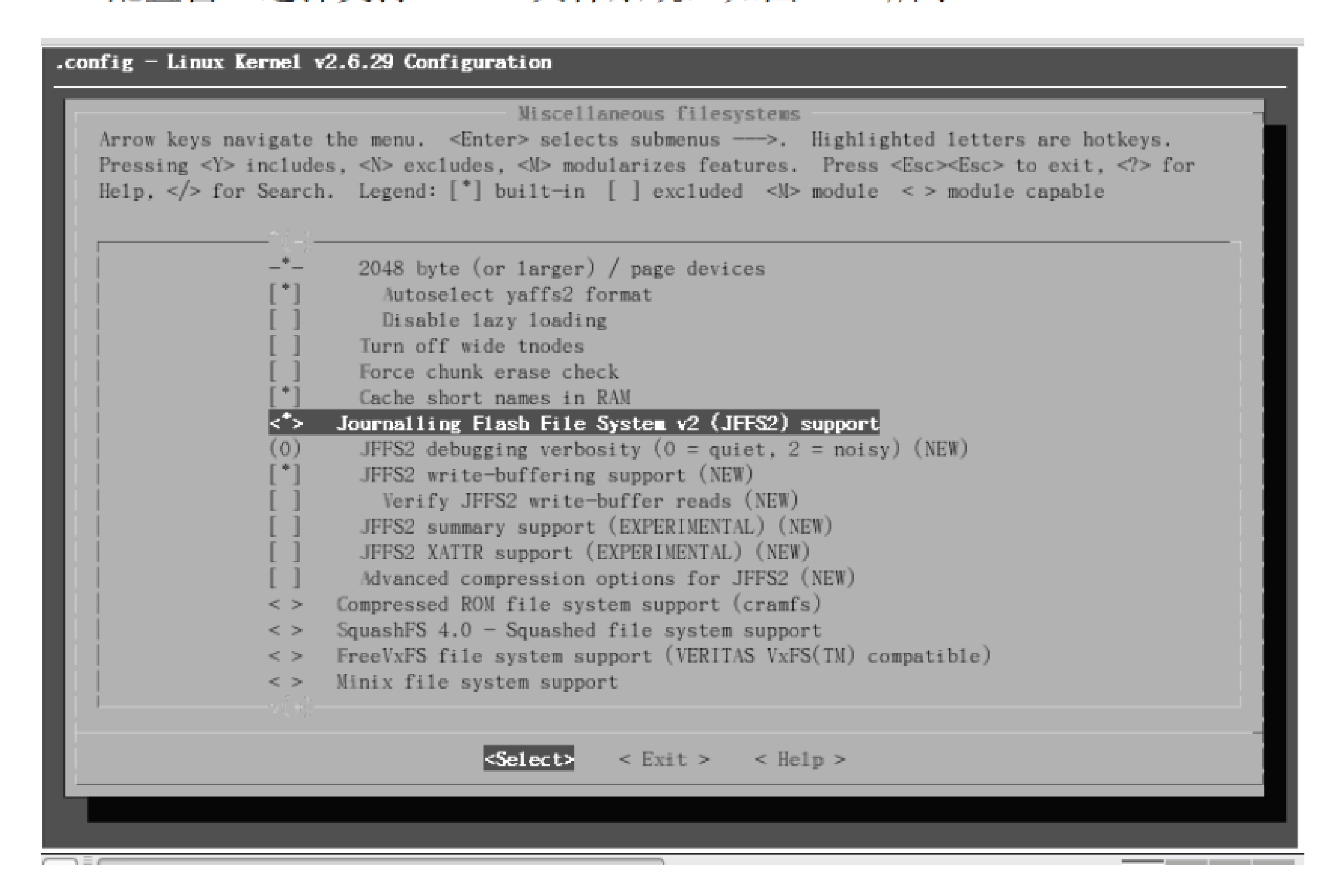



图 5.16　配置内核支持 JFFS2 文件系统

### 2．制作工具mkfs.jffs2

制作工具 mkfs.jffs2 是用于制作 JFFS2 映像文件。制作 JFFS2 映像文件需要以下两个文件：

- zlib-1.2.3.tar.gz；
- mtd-utils-1.0.0.tar.gz。

（1）编译安装 zlib 库，用于文件压缩和解压。进入 zlib 的解压目录下，使用 configure 命令生成 Makefile。

```
# ./configure --prefix=/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc
-shared
```

修改生成的 Makefile 如下：

```
CC=arm-linux-gcc
LDSHARED=arm-linux-gcc
CPP=arm-linux-gcc -E
AR=arm-linux-ar rc
RANLIB=arm-linux-ranlib
```

执行 make 和 make install 进行编译和安装。

```
#make
#make install
```

编译和安装完成后在目录/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib 下会生成动态和静态库文件 libz.a、libz.so、libz.so.1.2.3 和 libz.so.1。

（2）编译工具 mtd-utils。进入 mtd-utils 的解压目录，执行 make 进行编译。完成编译后，新生成的工具 mkfs.jffs2、mkfs.jffs 等在目录/usr/local/jffs_root/tmp/mtd-utils-1.0.0 下。

```
# tar zxvf mtd-utils-1.0.0.tar.gz
# cd mtd-utils-1.0.0
# make
```

将该路径添加到环境变量 PATH 中。

```
#PATH=$PATH: /usr/local/jffs_root/tmp/mtd-utils-1.0.0
```

（3）制作 JFFS2 映像文件。制作 JFFS2 根文件系统的过程与其他文件系统的过程相同，制作 JFFS2 映像文件的命令如下：

```
#mkfs.jffs2 -r jffs_root -o jffs_root.jffs2 -e 0x4000 --pad=0x800000 -s 0x200
-n
```

各个参数的含义如下所示。

- -r：指定文件系统。
- -o：指定输出的映像文件名。
- -e：擦除块的大小（block size），不同的 flash，其 block size 不一样。
- --pad （-p）：指定输出文件的大小，也就是 jffs_root.jffs2 的大小。重要的是，为了不浪费 flash 的空间，该值应该符合 flash driver 划分块的大小。
- -n：在每个擦除块中不添加 cleanmarker（消除标志）。

## 5.4.4　其他文件系统制作

Cramfs 文件系统目录的制作方法与其他文件系统目录的制作方法相同，其映像文件的制作如下：

```
# mkfs.cramfs rootfs cram.img
```

- rootfs：Cramfs 文件系统目录；
- cram.img：生成映像文件名。

Romfs 文件系统的制作一般使用工具 genromfs。下载 genromfs-0.5.1.tar.gz 进行解压，进入解压目录进行编译生成 genromfs 工具。

```
# tar zxvf genromfs-0.5.1.tar.gz
# cd genromfs-0.5.1
# make
```

使用 genromfs 工具制作 Cramfs 文件系统映像文件 romfs.img。使用如下命令可以看到制作 Cramfs 文件系统映像文件的细节。

```
# ./genromfs -V "xromfs" -f romfs.img -d ../rootfs/ -v
```

各个参数的含义如下所示。

- -V VOLUME：指定卷标；
- -f IMAGE：指定输出 romfs 映像的名字；
- -d DIRECTORY：指定源目录（将该目录制作成 romfs 文件系统）；
- -v：显示详细的创建过程。

## 5.5　小　　结

文件系统内容比较多，大致可以将其制作过程分为 3 个部分。工具程序，一般利用交叉编译 Busybox 获得；动态库文件，可以从交叉编译器的库目录下进行复制；配置和启动文件目录（etc），该目录下主要有四个文件，这四个文件可以参考生成的旧的文件系统进行配置。文件系统的内容虽然很多，制作文件系统最简单有效的方式还是在原有的文件系统目录基础上进行增加和删除。

# 第 3 篇　系统移植与驱动篇

# 第 6 章　LCD 驱动移植

液晶显示器（Liquid Crystal Display，简称 LCD）是一种采用液晶控制透光度技术来实现色彩的显示器。它是嵌入式系统中常见的输出设备，也是现代掌上设备人机交互的重要组成部分。随着嵌入式技术的快速发展，LCD 在人们日常生活中可以说是无处不在，它在嵌入式产品中发挥着越来越重要的作用。本章中将讲述怎么在开发板上移植 Linux 的 LCD 驱动程序。

## 6.1　认识 LCD 相关硬件原理

在编写 LCD 驱动程序前，驱动开发者应该对 LCD 的相关硬件原理有个大概的认识，弄清楚 LCD 显示屏相关参数的意义，如何在驱动程序中设置这些参数，从而根据相应的 LCD 显示屏型号编写相应的驱动程序。本节首先对 LCD 显示屏做大概的讲述，在对 LCD 显示屏的显示原理有一个认识之后，主要讲述 S3C2440 芯片的 LCD 控制器。

### 6.1.1　LCD 概述

LCD（液晶显示）模块满足了嵌入式系统日益提高的要求。它可以显示汉字、字符和图形，同时还具有低压、低功耗、体积小、重量轻和超薄等很多优点。随着嵌入式系统的应用越来越广泛，功能也越来越强大，对系统中的人机界面的要求也越来越高。在实际应用的驱使下，许多工作在 Linux 下的图形界面软件包的开发和移植工作中都涉及底层 LCD 驱动的开发问题。因此开发 LCD 驱动在嵌入式系统中得以广泛运用。

#### 1．LCD显示屏的分类

常见的液晶显示屏按物理结构可分为 4 种，即扭曲向列型（TN-LCD）、超扭曲向列型（STN-LCD）、双层超扭曲向列型（DSTN-LCD）和薄膜晶体管型（TFT-LCD）。其中，TN-LCD、STN-LCD 和 DSTN-LCD 的基本显示原理是一样的，只是液晶分子扭曲的角度不同而已。而 TFT-LCD 则采用与 TN 型系列 LCD 完全不同的显示方式。在写驱动程序时要根据不同类型对 LCD 控制器进行控制。

#### 2．LCD的常用参数

市场上的 LCD 显示屏从厂家、型号、规格等来说不尽相同，了解 LCD 的主要参数对进行 LCD 驱动开发非常有用，因为写驱动程序时就需要对 LCD 的参数进行设置。

- ❑ PPI（Pixel Per Inch）是指每平方英寸所拥有的像素数目。由此可见，PPI 值越高就

意味着显示屏显示图像的密度越高，显示密度越高，拟真度也就越高，图像也就越清晰，显示效果也就越好。目前市面上通用的 TFT 液晶屏大部分是 100PPI 的。

- 分辨率：市面上的分辨率标准多种多样，主要有 VGA、SVGA、UXGA 和 SXGA+。其中，SXGA+所代表的显示分辨率为 1400×1050。Quad-VGA 是三菱公司的一种新分辨率标准，它所代表的分辨率是 1280×960，而一般标准 XGA 的代表是 1280×1024。
- BPP（Bit Per Pixel），即每个像素使用多少位来表示其颜色。比如，黑白色只用 1bit（1BPP）就可以表示黑白两种颜色，对于 4 阶灰度可用 2bit（2BPP）来表示一个点，而对于 256 色的彩色要用 8bit（8BPP）来表示一个点。

#### 3．LCD的显示原理

对于内存中的一幅完整图像，它是怎么显示到屏幕上的呢？LCD 显示器沿用了以前 CRT 显示器的概念。一幅图像被称为一帧，每帧由多个行排列组成，每行又由多个像素组成，每个像素的色彩使用若干位数据来表示。对于单色（黑白）显示器，每个像素用 1 位来表示，称为 1BPP；对于 256 色显示器，每 个像素使用 8 位来表示，称为 8BPP，依此类推。

显示器从屏幕的左上方开始，一行一行地取得每个像素的数据并显示出来，当显示到一行的最右边时，跳到下一行的最左边开始显示下一行；当显示完所有行后，重新跳到左上方开始下一帧的显示。显示器沿着“Z”字形的路线进行扫描，同时使用帧扫描信号和行扫描信号来同步每一帧和每一行。

### 6.1.2　LCD 控制器

LCD 控制器的功能是产生控制时序和信号，从而驱动 LCD。用户只需要通过读写 LCD 控制器的一系列寄存器来完成配置。用户所要显示的内容皆是由 LCD 控制器从帧缓冲区中读出，然后再把读取到的数据发送到 LCD 驱动器进而显示到屏幕上。随着电子技术不断的发展，芯片的集成度也越来越高了，很多嵌入式处理器都集成了 LCD 控制器，如三星的 S3C2440，本节将以 S3C2440 的 LCD 控制器为例来分析如何设置 LCD 控制寄存器。

S3C2440 的 LCD 控制器由一个逻辑单元组成，它的作用是把 LCD 图像数据从一个位于系统内存的 buffer 传送到一个外部的 LCD 驱动器。LCD 控制器使用一个基于时间的像素抖动算法和侦速率控制思想，可以支持单色，2-bit per pixel（4 级灰度）或者 4-bit-pixel（16 级灰度）屏，并且它可以与 256 色（8BPP）和 4096 色（12BPP）的彩色 STN-LCD 连接。

LCD 控制器还支持 1BPP、2BPP、4BPP、8BPP 的调色板 TFT 彩色屏，并且支持 64K 色（16BPP）和 16M 色（24BPP）非调色板真彩显示。LCD 控制器可以编程满足不同的需求，如水平或者垂直方向的像素数目，数据接口的数据线宽度，接口时序和刷新速率等。

### 6.1.3　LCD 控制器方块图

图 6.1 描述了 S3C2440 的 LCD 控制器结构。由于工作原理不同，LCD 控制器的接口

时序分为 STN 和 TFT 两种。S3C2440 的 LCD 控制器可以同时支持 STN 和 TFT 的 LCD 显示屏，根据实际需要对控制器进行不同的设置可以产生不同的时序。

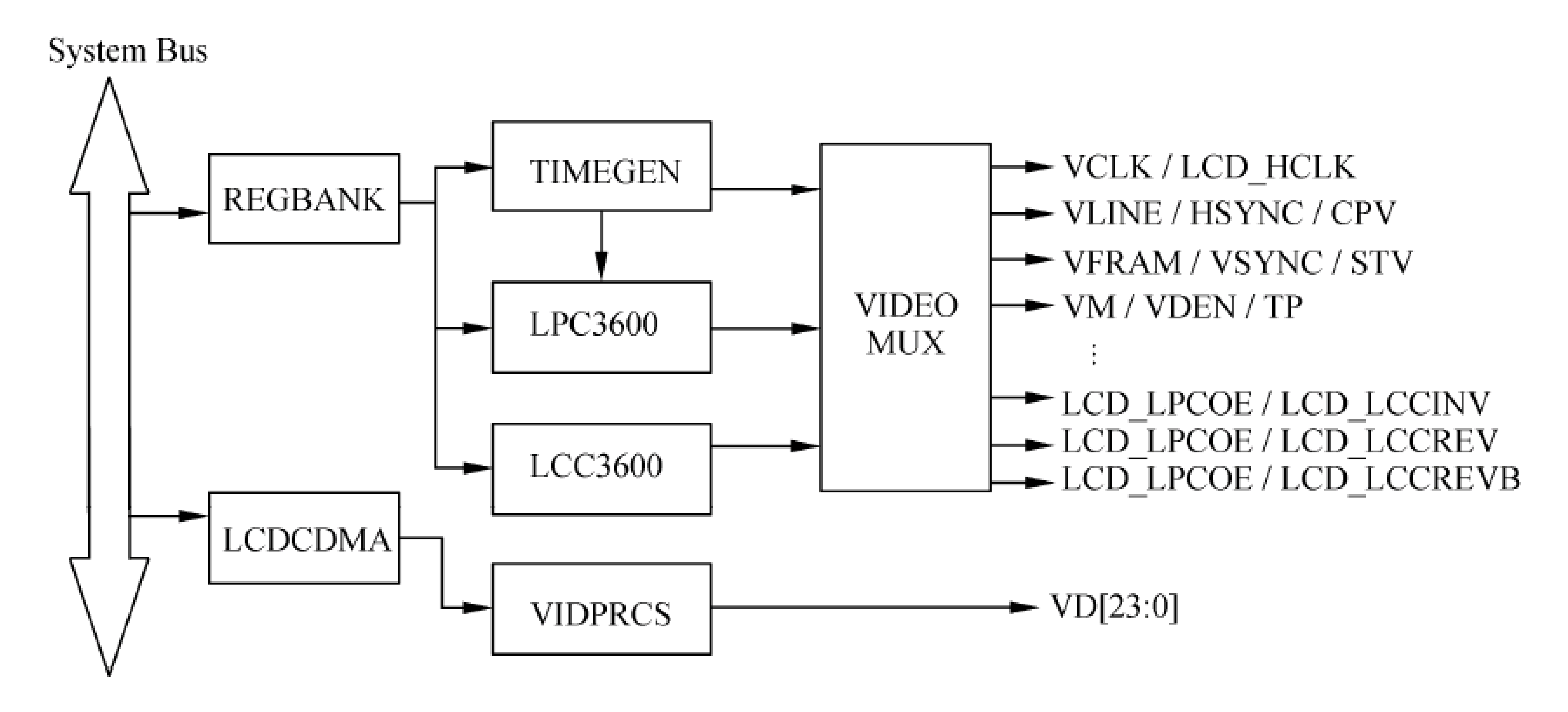



图 6.1　S3C2440 的 LCD 控制器结构

S3C2440 LCD 控制器被用来传送视频数据和生成必要的控制信号，比如 VFRAME、VLINE、VCLK 和 VM 等。除了控制信号外，S3C2440 还有作为视频数据的数据端口，它们是如图 6.1 所示的 VD[23:0]。LCD 控制器由 REGBANK、LCDCDMA、VIDPRCS、TIMEGEN 和 LPC3600 组成。

REGBANK 由 17 个可编程的寄存器组和一块 256×16 的调色板内存组成，这些是用来配置 LCD 控制器的。LCDCDMA 是一个专用的 DMA，它能自动地把在内存中的视频数据传送到 LCD 驱动器。通过使用这个 DMA 通道，视频数据在不需要 CPU 干预的情况下显示在 LCD 屏上。VIDPRCS 接收从 LCDCDMA 到来的数据，将数据转换为合适的数据格式，比如说 4/8 位单扫，4 位双扫显示模式，然后通过数据端口 VD[23:0]传送视频数据到 LCD 驱动器。TIMEGEN 由可编程的逻辑组成，支持不同的 LCD 驱动器接口时序和速率的需要，TIMEGEN 块可以产生 VFRAME、VLINE、VCLK 和 VM 等时钟信号。

具体的数据流描述如下：

LCDCDMA 中存在 FIFO 存储器。当 FIFO 为空，或者部分为空的时候，LCDCDMA 请求从存储器中取得数据，是用突发的存储传输模式取得数据的（每一个突发请求，连续的取 4 个字（16bytes），在总线传输过程中，不允许总线控制权交给另一个总线控制）。当传输请求被存储控制器中的总线仲裁器接收后，将会产生连续的 4 个字的数据传输从系统内存到内部的 FIFO。FIFO 的总共大小为 28 个字，由 12 个字的 FIFOL 和 16 个字的 FIFOH 分别组成。S3C2440 用 2 个 FIFO 支持双扫显示模式。假如是单扫模式，只有一个 FIFO 会被用到。

## 6.1.4　LCD 控制器操作

S3C2440 的 LCD 控制器分 STN 控制和 TFT 控制，目前市面上主流的 LCD 为 TFT-LCD，因此本节将基于 TFT-LCD 介绍 LCD 控制器的使用。对于 STN-LCD 所涉及的操作与此

类似。

首先来了解一下视频数据在内存中是怎么存储的。

显示屏上每个像素的色彩由 3 个部分组成，即红、绿、蓝，这就是我们经常说的三基色。这 3 种颜色混合可以表示人眼所能识别的几乎所有颜色。如果每种颜色用 8bit 来表示，则每种颜色可分为 256 阶色，那么一个像素点就可以用 24bit（24BPP）来表示，每个像素就有 16M 阶。S3C2440 的 TFT LCD 控制器支持 1、2、4、8BPP 调色板彩色模式及 16BPP、24BPP 无调色板真彩模式。下面分别介绍各显示模式下图像数据的存储格式。

### 1．24BPP显示

24BPP 显示模式使用 24 位表示一个像素点，每种颜色用 8 比特位来表示。LCD 控制器从内存中获得某个像素的 24 位颜色值后直接通过 VD[23:0]数据线发送给 LCD 驱动器，像素值与 VD[23:0]引脚的对应关系如表 6.1 所示。

**表 6.1　像素值与VD[23:0]引脚的对应关系**

| VD | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RED | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | | | | | | | | | | |
| GREEN | | | | | | | | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | | |
| BLUE | | | | | | | | | | | | | | | | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

为了方便 DMA 传输，在内存中使用 4 字节来表示一个像素，其中有一位是无效的。可以通过编程来选择最低字节无效还是最高字节无效。内存中像素的排列格式，分两种情况，高位无效和低位无效，如表 6.2 和表 6.3 所示，图 6.2 为像素在屏幕上的排列情况。

**表 6.2　24BPP内存中像素的排列格式（BSWP=0，SHSWP=0，BPP24BL=0）**

| | D[31:24] | D[23:0] |
|---|---|---|
| 000H | Dummy Bit | P1 |
| 004H | Dummy Bit | P2 |
| 008H | Dummy Bit | P3 |
| ... | | |

**表 6.3　24BPP内存中像素的排列格式（BSWP=0，SHSWP=0，BPP24BL=1）**

| | D[31:8] | D[7:0] |
|---|---|---|
| 000H | P1 | Dummy Bit |
| 004H | P2 | Dummy Bit |
| 008H | P3 | Dummy Bit |
| ... | | |

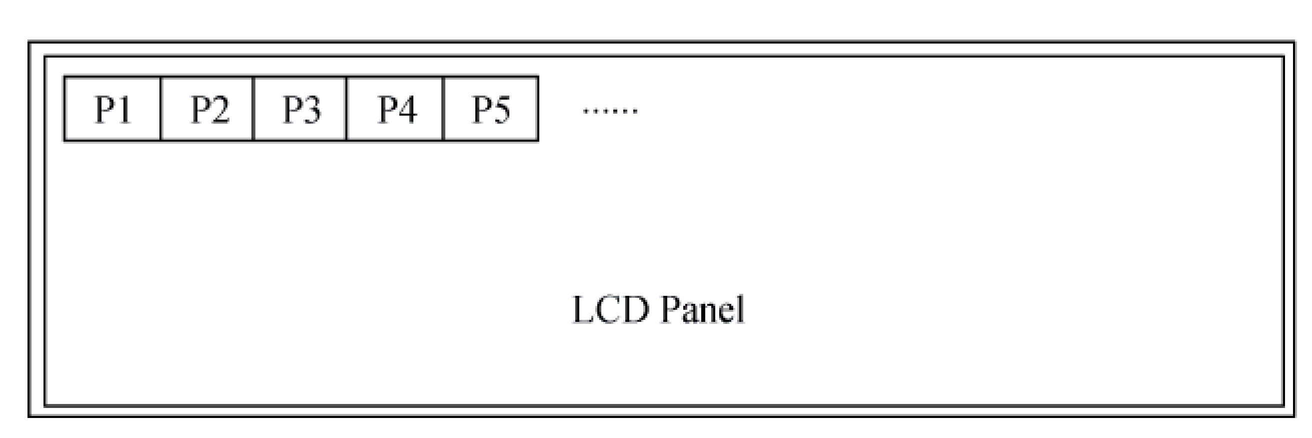



图 6.2　24BPP 显示模式时像素在屏幕上的排列情况

### 2. 16BPP

16BPP 模式用 16 位来表示一个像素点的值。这 16 位数据的格式分为两种：5:6:5 和 5:5:5:1，前者使用高 5 位表示红色，中间 6 位表示绿色，低 5 位表示蓝色；后者用高 15 位表示红、绿、蓝 3 种颜色，每种颜色用 5 位表示，最低位表示透明度。

内存数据与像素位置的关系如表 6.4 所示。NC 表示没有连接；5:5:5:1 格式下“I”表示透明度。

**表 6.4　16BBP像素值与VD[23:0]引脚的对应关系**

（5:6:5）

| VD | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RED | 4 | 3 | 2 | 1 | 0 | NC | | | | | | | | | NC | | | | | | | NC | | |
| GREEN | | | | | | | | | 5 | 4 | 3 | 2 | 1 | 0 | | | | | | | | | | |
| BLUE | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | | | |

（5:5:5:1）

| VD | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RED | 4 | 3 | 2 | 1 | 0 | I | NC | | | | | | | | NC | | | | | | | | NC | |
| GREEN | | | | | | | | | 4 | 3 | 2 | 1 | 0 | I | | | | | | | | | | |
| BLUE | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | I | | |

4 字节可以表示两个像素，使用高 2 字节还是低 2 字节来表示第一个像素也是可以编程选择的，如表 6.5 和表 6.6 所示，图 6.3 为 16BPP 模式时像素在屏幕上的排列情况。

**表 6.5　16BPP内存中像素的排列格式（BSWP=0，SHSWP=0）**

| | D[31:16] | D[15:0] |
|---|---|---|
| 000H | P1 | P2 |
| 004H | P3 | P4 |
| 008H | P5 | P6 |
| … | | |

**表 6.6　16BPP内存中像素的排列格式（BSWP=0，SHSWP=1）**

| | D[31:16] | D[15:0] |
|---|---|---|
| 000H | P2 | P1 |
| 004H | P4 | P3 |
| 008H | P6 | P5 |
| … | | |

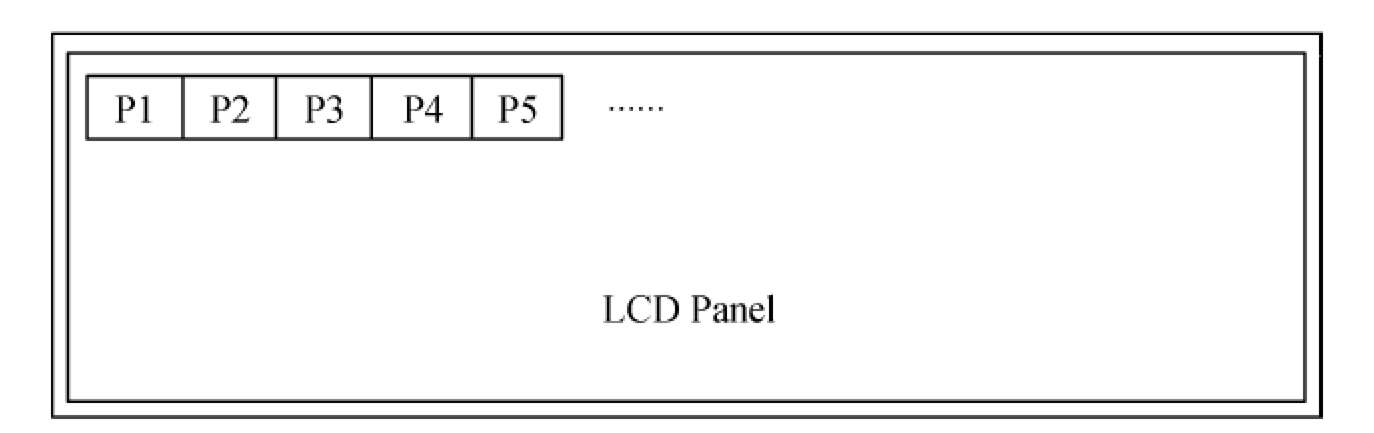



图 6.3　16BPP 显示模式时像素在屏幕上的排列情况

### 3．8BPP

8BPP 显示模式使用 8 位来表示一个像素点，然而对 3 种基色平均下来，每种基色只能使用不到 3 位的数据来表示，即每种基色最多不过 8 阶，这不能表示更丰富的色彩。

4 字节可以表示 4 个 8BPP 的像素，字节与像素的对应顺序也是可以编程选择的，如表 6.7 和表 6.8 所示，图 6.4 为 8BPP 模式时像素在屏幕上的排列情况。

**表 6.7　8BPP内存中像素的排列格式（BSWP=0，SHSWP=0）**

| | D[31:24] | D[23:16] | D[15:8] | D[7:0] |
|---|---|---|---|---|
| 000H | P1 | P2 | P3 | P4 |
| 004H | P5 | P6 | P7 | P8 |
| 008H | P9 | P10 | P11 | P12 |
| … | | | | |

**表 6.8　8BPP内存中像素的排列格式（BSWP=0，SHSWP=1）**

| | D[31:24] | D[23:16] | D[15:8] | D[7:0] |
|---|---|---|---|---|
| 000H | P4 | P3 | P2 | P1 |
| 004H | P8 | P7 | P6 | P5 |
| 008H | P12 | P11 | P10 | P9 |
| … | | | | |

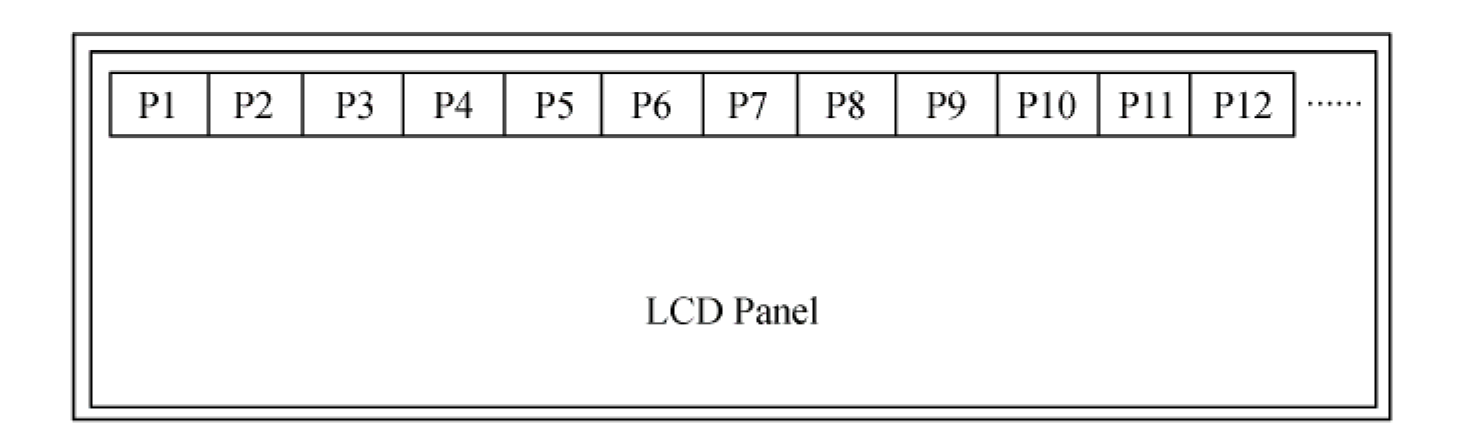



图 6.4　8BPP 显示模式时像素在屏幕上的排列情况

### 4．256色调色板

为了解决 8BPP 模式下显示能力弱的问题，需要使用调色板。每个像素对应的 8 位数据不是用来表示 RGB 3 种基色，而是表示它在调色板中的索引值；要显示这个像素时，使用这个索引值从调色板中取得其 RGB 值。这里说的调色一块内存，可以对每个索引值设置颜色，可以使用 24BPP 或 16BPP。S3C2440 为 TFT 显示器提供 256 色调色板，调色板是一块 256×16 的内存，使用 16BPP 的格式来表示 8BPP 模式下各索引值的颜色。8BPP 模式下每种基色只能使用不到 3 位，所以每种基色最大的索引值为 8，只能表示 256 阶中的 8 个阶值。这样，即使使用 8BPP 的显示模式，最终出现在 LCD 数据总线上的仍是 16BPP 的数据。

调色板中数据存放的格式与上面所描述的 16BPP 显示模式相似，也分两种格式：5:6:5 和 5:5:5:1。调色板中数据的格式及其与 LCD 数据线的对应关系如表 6.9 所示。

表 6.9　调色板中数据的格式及其与LCD数据线的对应关系

5:6:5 格式：

| INDEX\Bit Pos | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Address |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00H | R4 | R3 | R2 | R1 | R0 | G5 | G4 | G3 | G2 | G1 | G0 | B4 | B3 | B2 | B1 | B0 | 0X4D000400 |
| 01H | R4 | R3 | R2 | R1 | R0 | G5 | G4 | G3 | G2 | G1 | G0 | B4 | B3 | B2 | B1 | B0 | 0X4D000404 |
| … | | | | | | | | | | | | | | | | | |
| FFH | R4 | R3 | R2 | R1 | R0 | G5 | G4 | G3 | G2 | G1 | G0 | B4 | B3 | B2 | B1 | B0 | 0X4D0007FC |
| Number of DV | 23 | 22 | 21 | 20 | 19 | 15 | 14 | 13 | 12 | 11 | 10 | 7 | 6 | 5 | 4 | 3 | |

5:5:5:1 格式：

| INDEX\Bit Pos | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Address |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00H | R4 | R3 | R2 | R1 | R0 | G5 | G4 | G3 | G2 | G1 | G0 | B4 | B3 | B2 | B1 | B0 | 0X4D000400 |
| 01H | R4 | R3 | R2 | R1 | R0 | G5 | G4 | G3 | G2 | G1 | G0 | B4 | B3 | B2 | B1 | B0 | 0X4D000404 |
| … | | | | | | | | | | | | | | | | | |
| FFH | R4 | R3 | R2 | R1 | R0 | G5 | G4 | G3 | G2 | G1 | G0 | B4 | B3 | B2 | B1 | B0 | 0X4D0007FC |
| Number of DV | 23 | 22 | 21 | 20 | 19 | 15 | 14 | 13 | 12 | 11 | 10 | 7 | 6 | 5 | 4 | 3 | |

**说明：** 0x4D000400 是调色板的起始地址；VD18、VD10 和 VD2 有同样的输出值；.DATA[31:16]是无效的。

现在，读者基本知道视频数据在内存中的存储格式了，下面再来了解一下这些数据是如何显示到 LCD 屏的，这就要了解 LCD 的时序，下面开始介绍 TFT-LCD 时序。

TFT-LCD 时序图如图 6.5 所示。

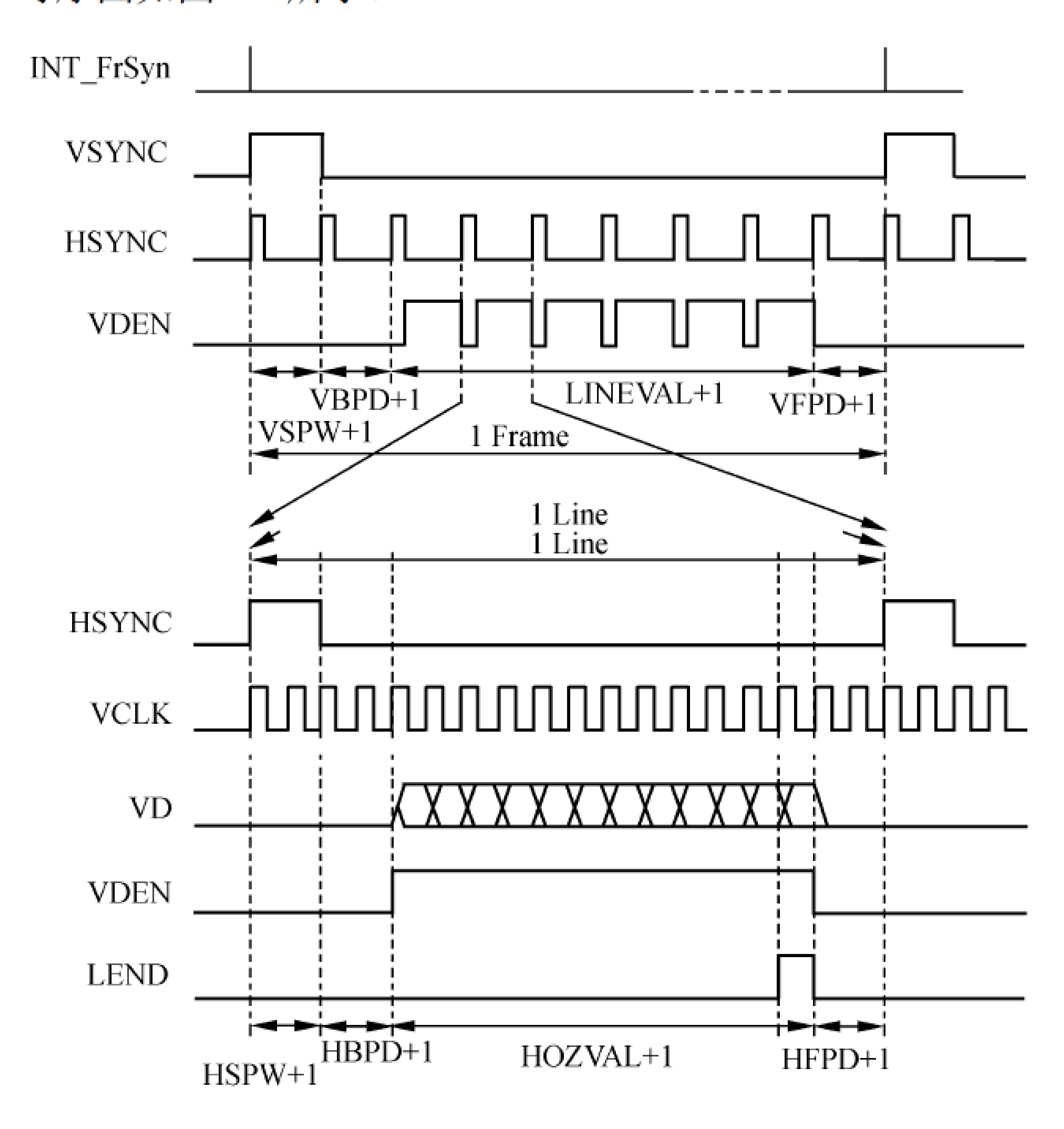



图 6.5　TFT-LCD 时序图

图 6.5 是 TFT 屏的典型时序。其中 VSYNC 是帧同步信号，VSYNC 每发出 1 个脉冲，都意味着新的 1 屏视频资料开始发送。而 HSYNC 为行同步信号，每个 HSYNC 脉冲都表明新的 1 行视频资料开始发送。而 VDEN 则用来标明视频资料的有效，VCLK 是用来锁存视频资料的像素时钟。

在帧同步及行同步的头尾都必须留有回扫时间，例如对于 VSYNC 来说，前回扫时间就是（VSPW+1）+（VBPD+1），后回扫时间就是（VFPD+1）；HSYNC 亦类同。这样的时序要求是当初 CRT 显示器由于电子枪偏转需要时间，但后来成了实际上的工业标准，乃至于后来出现的 TFT 屏为了在时序上与 CRT 兼容，也采用了这样的控制时序。

## 6.1.5　LCD 控制寄存器

S3C2440 提供了 LCD 控制器，其中有 17 个控制寄存器，包括 LCDCON1～LCDCON5 及 LCDSADDR1- LCDSADDR3 等，下面将分别介绍。

### 1．LCD控制寄存器LCDCON1

LCDCON1 用于选择 LCD 类型、设置像素时钟、使能 LCD 输出信号等，其格式如表 6.10 所示。

表 6.10　LCDCON1 控制格式

| 功　　能 | 位 | 描　　述 | 初始状态 |
|---|---|---|---|
| LINECNT（只读） | [27:18] | 每输出一个有效行其值减 1 | 0 |
| CLKVAL | [17:8] | 用于设置 VCLK 的值<br>STN：VCLK=HCLK/（CLKVAL×2）<br>TFT：VCLK=HCLK/（（CLKVAL+1）×2） | 0 |
| MMODE | [7] | 用于设置 VM 信号的反转效率，只用于 TFT 屏 | 0 |
| PNRMODE | [6:5] | 设置 LCD 的类型<br>00：4 位双扫描（STN）<br>01：4 位单扫描（STN）<br>10：8 位单扫描（STN）<br>11：TFT 屏 | 0 |
| BPPMODE | [4:1] | 选择 BPP 模式，对于 TFT 屏<br>1000：1BPP<br>1001：2BPP<br>1010：4BPP<br>1011：8BPP<br>1100：16BPP<br>1101：24BPP（STN） | 0 |
| ENVID | [0] | 信号输出使能位<br>0：禁止，1：使能 | 0 |

### 2．LCD控制寄存器LCDCON2

用于设置垂直方向各信号的时间参数，其格式如表 6.11 所示。

表 6.11　LCDCON2 控制格式

| 功　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| VBPD | [31:24] | VSYNC 信号之后，还要经过（VBPD+1）个 HSYNC 信号周期才会出现有效行数据 | 0 |
| LINEVAL | [23:14] | LCD 的行数：（LINEVAL+1）行 | 0 |
| VFPD | [13:6] | 一帧中的有效数据完成后，到下一个 VSYNC 信号有效前的无效行数目：（VFPD+1） | 0 |
| VSPW | [5:0] | 表示 VSYNC 信号的宽度为（VSPW+1）个 HSYNC 信号周期，这（VSPW+1）行的数据无效 | 0 |

### 3．LCD控制寄存器LCDCON3

用于设置水平方向各信号的时间参数，其格式如表 6.12 所示。

表 6.12　LCDCON3 控制格式

| 功　　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| HBPD | [25:19] | HSYNC 信号脉冲之后，还要经过（HBPD+1）个 VCLK 信号周期，才出现有效数据 | 0 |
| HOZVAL | [18:8] | LCD 的水平宽度：（HOZVAL+1）个像素 | 0 |
| HFPD | [7:0] | 一行中的有效数据完后，到下一个 HSYNC 信号有效前的无效像素个数：HFPK+1 | 0 |

### 4．LCD控制寄存器LCDCON4

对于 TFT 屏，这个寄存器用来设置 HSYNC 信号的脉冲宽度，其格式如表 6.13 所示。

表 6.13　LCDCON4 控制格式

| 功　　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| MVAL | ［15:8］ | STN 屏用 | 0 |
| HSPW | [7:0] | 表示脉冲宽度为（HSPW+1）个 VCLK 信号周期 | 0 |
| WLH | ［7:0］ | STN 屏用 | 0 |

### 5．LCD控制寄存器LCDCON5

用于设置各个控制信号的极性，并可从中读取状态信息，其格式如表 6.14 所示。

表 6.14　LCDCON5 控制格式

| 功　　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| VSTATUS | [16:15] | 只读，垂直状态<br>00：处于 VSYNC 信号脉冲期间<br>01：处于 VSYNC 信号结束到行有效之间<br>10：处于行有效期间<br>11：处于行有效结束到下一个 VSYNC 信号之间 | 0 |

续表

| 功　　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| HSTATUS | [14:13] | 只读，描述水平状态<br>00：处于 HSYNC 信号脉冲期间<br>01：处于 HSYNC 信号结束到像素有效之间<br>10：处于像素有效期间<br>11：处于像素有效结束到下一个 HSYNC 信号之间 | 0 |
| BPP24BL | [12] | 设置 TFT 屏的显示模式为 24BPP 时，一个 4 字节中哪 3 个字节有效。<br>0：低 3 字节有效；1：高 3 字节有效 | 0 |
| FRM565 | [11] | 设置 TFT 屏的显示模式为 16BPP 时，数据的格式<br>0：5:5:5:1；1：5:6:5 | 0 |
| INVVCLK | [10] | 设置 VCLK 信号有效沿的极性<br>0：下降沿读取数据；1：在上升沿读取数据 | 0 |
| INVVLINE | [9] | 设置 VLINE/HSYNC 脉冲的极性<br>0：正常极性；1：反转的极性 | 0 |
| INVVFRAME | [8] | 设置 VFRAME/VSYNC 脉冲的极性<br>0：正常极性；1：反转的极性 | 0 |
| INVD | [7] | 设置 VD 数据线表示数据的极性<br>0：正常极性；1：反转的极性 | 0 |
| INVVDEN | [6] | 设置 VDEN 信号的极性<br>0：正常极性；1：反转的极性 | 0 |
| INVPWREN | [5] | 设置 PWREN 信号的极性<br>0：正常极性；1：反转的极性 | 0 |
| INVLEND | [4] | 设置 LEND 信号的极性<br>0：正常极性；1：反转的极性 | 0 |
| PWREN | [3] | LCD_PWREN 信号输出使能<br>0：禁止；1：使能 | 0 |
| ENLEND | [2] | LEND 信号输出使能<br>0：禁止；1：使能 | 0 |
| BSWP | [1] | 字节交换使能<br>0：禁止；1：使能 | 0 |
| HWSWP | [0] | 半字交换使能<br>0：禁止；1：使能 | 0 |

### 6．帧内存地址寄存器

帧内存可以很大，而真正要显示的区域称为视口，它处于帧内存之内。s3c2440 有 3 个帧内存地址寄存器，这 3 个寄存器用于确定帧内存的起始地址，定位视口在帧内存的位置，其格式如表 6.15 所示。

表 6.15　帧内存地址寄存器格式

LCDSADDR1

| 功　　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| LCDBANK | [29:21] | 用于保存帧内存起始地址的 A［30：22］，帧内存起始地址必须为 4M 对齐 | 0 |
| LCDBASEU | [20:0] | 对于双扫描，用于表示上半帧的内存起始地址<br>对于单扫描，用于表示帧的内存起始地址 | 0 |

LCDSADDR2

| 功　　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| LCDBASEL | [20：0] | 对于双扫描，用于保存下半帧的起始地址<br>对于单扫描，用于保存帧的结束地址<br>其值按如下公式计算：<br>LCDBASEL=CLDBASEU+（PAGEWIDTH+OFFSIZE）*（LINEVAL+1） | 0 |

LCDSADDR3

| 功　　能 | 位 | 描　　述 | 初始值 |
|---|---|---|---|
| OFFSIZE | [21:11] | 虚拟屏长度<br>表示上一行最后一个数据与下一行第一个数据间地址差值的一半，即以半字为单位的地址差 | 0 |
| PAGEWIDTH | [10:0] | 虚拟屏的宽度，这个值决定视口的宽度，以半字为单位 | 0 |

## 6.2　LCD 参数设置

LCD 驱动编写的主要任务就是根据所使用的 LCD 屏正确地设置对应的 LCD 寄存器参数。前面已经讲述了 S3C2440 LCD 各控制寄存器，本节中，我们将讲述如何设置这些寄存器参数。

### 1．设置VFRAME、VLINE

VFRAME 和 VLINE 信号可以根据液晶屏的尺寸和显示模式来设置，它们对应 LCDCON2 寄存器的 HOZVAL 和 LINEVAL 值，设置方法如下：

```
HOZVAL=（水平尺寸/VD 数据位）-1
彩色液晶屏：水平尺寸=3 × 水平像素点数
VD 数据位=BBP 数（不分单双扫描）
LINVAL=垂直尺寸-1（单扫描）
LINVAL=垂直尺寸/2-1（双扫描）
```

### 2．设置VCLK

LCD 控制器输出的 VCLK 是直接由系统总线（AHB）的工作频率 HCLK 直接分频得到的。

```
VCLK=HCLK/（（CLKVAL+1）× 2）
```

3．帧速率

帧速率就是 VSYNC 信号的频率。帧速率与 VSYNC、VBPD、VFPD、LINEVAL、HSYNC、HBPD、HFPD、HOZVAL 和 CLKVAL 的域有关，它们是 LCDCON1/2/3/4。大多数 LCD 驱动器需要它们合适的帧速率。帧速率按如下公式计算：

```
Frame Rate=1/[{(VSPW+1)+(VBPD+1)+(LINEVAL+1)+(VFPD+1)} *
{(HSPW+1)+(HBPD+1)+(HFPD+1)+HOZVAL+1}*{2*(CLKVAL+1)/HCLK}]
```

其中，VSPW、VBPD、VFPD、HSPW、HBPD、HFPD 需要参考具体的 LCD 屏的手册（datasheet）来设置。

# 6.3　内核 LCD 驱动机制

本节开始，将介绍如何编写 LCD 驱动程序。实际上，Linux 已经为显示设备专门提供了一类驱动程序，叫做帧缓冲（FrameBuffer）设备驱动程序。实际工作中，工程师只需要在显示缓存中填写将要显示的数据，屏幕上就会显示出相应的图像，驱动的主要工作就是准确地得到这个显示缓存的地址，然后对它进行操作。

## 6.3.1　FrameBuffer 概述

帧缓冲区是出现在 Linux 2.2.xx 及以后版本内核中的一种驱动程序接口，这种接口把显示设备抽象成为帧缓冲区设备区。帧缓冲区为图像设备提供了一种抽象化的处理，它代表了一些视频设备，允许应用软件通过定义明确的界面来访问图像硬件设备。这样软件无须了解任何涉及硬件底层驱动的东西（如硬件寄存器）。

帧缓冲区允许上层应用程序在图形模式下直接对显示缓冲区进行读写和 I/O 控制等操作。通过专门的设备结点可对相应的设备进行访问，如/dev/fb0。应用程序可以将它看成是显示内存的一个引用，将其映射到进程地址空间之后，就可以进行读写操作，而读写操作可以反映到具体的 LCD 设备上。

此外，FrameBuffer 驱动程序还考虑了支持控制台的字符显示。在 Linux 2.4 内核中，与 FrameBuffer 控制台有关的代码被放到了 fbcon 和其他相关目录中；在 Linux 2.6 内核中，这些代码被放到了 drivers/video/console 中，它们涵盖了各种格式显示缓冲的字符输出、字体定义文件，这样可以简化 FrameBuffer 控制台驱动程序的移植。

## 6.3.2　FrameBuffer 设备驱动的结构

FrameBuffer 设备驱动基于两个文件，linux/include/linux/fb.h 和 linux/drivers/video/fbmem.c。其中，fb.h 定义了 Framebuffer 驱动所要用到的几乎所有的结构体，这些结构主要包括 struct fb_info、struct fb_var_screeninfo 和 struct fb_fix_screeninfo。下面分别讲述这

几个结构体。

### 1．struct fb_info

struct fb_info 记录了帧缓冲的全部信息，包括设置参数、状态、操作函数指针等。每个帧缓冲设备均由一个 struct fb_info 体来描述它，struct fb_info 所包含的参数、状态、操作函数指针都是面向特定的设备，驱动开发工程师的任务就是填写这些数值。这个结构体也是所有 FrameBuffer 相关结构中唯一一个在内核空间可见的，具体结构代码如下：

```
struct fb_info {
    int node;                           /*次设备号，如 fb0 中的“0”*/
    int flags;
    struct fb_var_screeninfo var;       /*当前可变参数*/
    struct fb_fix_screeninfo fix;       /*当前固定参数*/
    struct fb_monspecs monspecs;        /*当前的显示器模式*/
    struct work_struct queue;           /*事件队列*/
    struct fb_pixmap pixmap;
    struct fb_pixmap sprite;
    struct fb_cmap cmap;                /*当前 cmap*/
    struct list_head modelist;          /*模式列表*/
    struct fb_videomode *mode;          /*当前模式*/
    struct fb_ops *fbops;               /*一些操作指针集，下面会讲到*/
    struct device *device;              /*指向 struct platform_device 中的
                                        dev 成员*/
    struct class_device *class_device;  /*sysfs 文件系统用到*/
#ifdef CONFIG_FB_TILEBLITTING
    struct fb_tile_ops *tileops;        /*Tile Blitting*/
#endif
    char __iomem *screen_base;          /*硬件 I/O 的虚拟地址*/
    unsigned long screen_size;
    void *pseudo_palette;               /*调色板*/
#define FBINFO_STATE_RUNNING   0
#define FBINFO_STATE_SUSPENDED 1
    u32 state;                          /*硬件状态*/
    void *fbcon_par;
    /*From here on everything is device dependent*/
    void *par;                          /*驱动定义的私有数据*/
};
```

由上可见，struct fb_var_screeninfo 和 struct fb_fix_screeninfo 两个结构体被包含于 struct fb_info，这两个结构记录了设备状态信息。它们和 struct fb_info 的关系如图 6.6 所示。

由图可以看出 struct fb_var_screeninfo 与 fb_fix_screeninfo 的区别，struct fb_var_screeninfo 可以通过 fb_ops 接口调用进行参数的设置。fb_ops 定义了一些操作函数，用户应用程序可以使用 ioctl()系统调用来操作设备，这个结构就是用于支持 ioctl()的这些操作。

### 2．fb_var_screeninfo

struct fb_var_screeninfo 记录了帧缓冲设备和指定显示模式的可修改信息。它包括显示屏幕的分辨率、每个像素的比特数和一些时序变量。其中变量 xres 定义了屏幕中一行所占像素的数目，yres 定义了屏幕中一列所占像素的数目，bits_per_pixel 定义了每个像素用多

少个位来表示。这些都是可以通过应用程序来设置的，它通过 fb_opt 实现。

fb_info

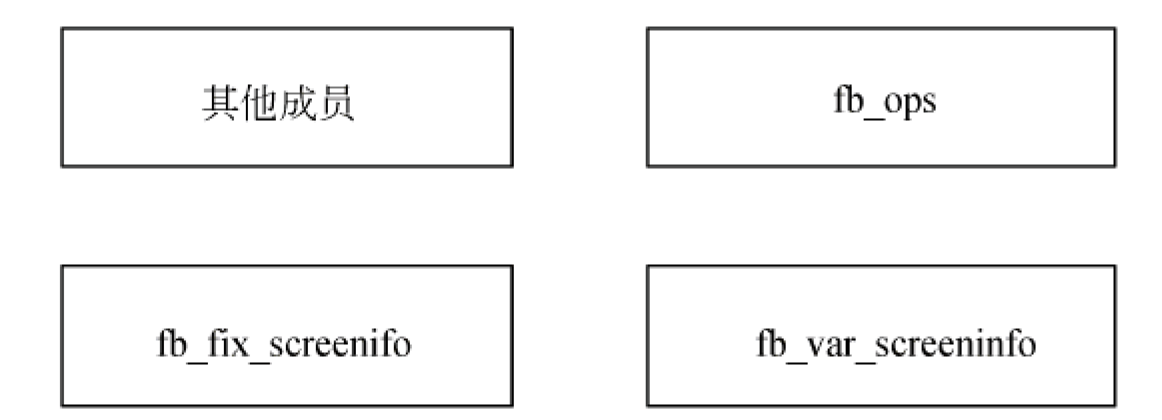



图 6.6　fb_info 结构图

```
struct fb_var_screeninfo {
    __u32 xres;                         /*实际分辨率*/
    __u32 yres;
    __u32 xres_virtual;                 /*虚拟分辨率*/
    __u32 yres_virtual;
    __u32 xoffset;                      /*虚拟分辨率到实际分辨率的偏移*/
    __u32 yoffset;

    __u32 bits_per_pixel;               /*BPP*/
    __u32 grayscale;                    /*灰度级别*/

    struct fb_bitfield red;             /*真彩色中的三基色*/
    struct fb_bitfield green;
    struct fb_bitfield blue;
    struct fb_bitfield transp;

    __u32 nonstd;                       /*!= 0 不是超标准格式*/

    __u32 activate;

    __u32 height;                       /*图像高度*/
    __u32 width;                        /*图像宽度*/

    __u32 accel_flags;                  /*加速标志*/

    /*下面是一些时序*/
    __u32 pixclock;                     /*像素时钟*/
    __u32 left_margin;                  /*帧的同步时钟    */
    __u32 right_margin;
    __u32 upper_margin;                 /*行的同步时钟    */
    __u32 lower_margin;
    __u32 hsync_len;                    /*一行的像素数    */
    __u32 vsync_len;                    /*一帧的行数*/
    __u32 sync;
    __u32 vmode;
    __u32 rotate;
    __u32 reserved[5];
};
```

### 3．fb_fix_screeninfo

而 fb_fix_screeninfo 定义了硬件的不可变属性，显示缓冲区的映射地址也被定义在这里，它表示缓冲区不应该被应用程序所改变。下面是这个结构体的代码：

```
struct fb_fix_screeninfo {
    char id[16];                    /*驱动中定义的设备名字*/
    unsigned long smem_start;       /*frame buffer 的开始地址，这是物理地址*/

    __u32 smem_len;                 /*frame buffer 内存的长度*/
    __u32 type;                     /*类型*/
    __u32 type_aux;                 /*Interleave for interleaved Planes*/
    __u32 visual;                   /*see FB_VISUAL_**/
    __u16 xpanstep;                 /*如果硬件没有 panning，那么填 0*/
    __u16 ypanstep;                 /*如果硬件没有 panning，那么填 0*/
    __u16 ywrapstep;                /*如果硬件没有 panning，那么填 0*/
    __u32 line_length;              /*一行的字节表示*/
    unsigned long mmio_start;       /*frame buffer 的开始地址，这是虚拟地址*/

    __u32 mmio_len;                 /*I/O 的大小*/
    __u32 accel;                    /*可用的加速类型  */

    __u16 reserved[3];              /*保留位*/
};
```

### 4．fbmem.c

fbmem.c 是 Framebuffer 设备驱动技术的关键。它为上层应用程序提供系统调用也为下一层的特定硬件驱动提供编程接口；那些底层硬件驱动需要用到这里的接口来向系统内核注册它们自己。

fbmem.c 为所有支持 FrameBuffer 的设备驱动提供了通用的接口，避免了重复的工作。用户开发硬件驱动时，只要针对 fbmem.c 中提供的底层驱动接口函数分别加以实现即可。这些接口函数就是前面讲到的 struct fb_ops 函数指针组，具体代码如下：

```
struct fb_ops {
    struct module *owner;
    int (*fb_open)(struct fb_info *info, int user);
    int (*fb_release)(struct fb_info *info, int user);
        ssize_t (*fb_read)(struct fb_info *info, char __user *buf,
                size_t count, loff_t *ppos);
    ssize_t (*fb_write)(struct fb_info *info, const char __user *buf,
                size_t count, loff_t *ppos);
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);
    int (*fb_set_par)(struct fb_info *info);
    int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
                unsigned blue, unsigned transp, struct fb_info *info);
    int (*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);
    int (*fb_blank)(int blank, struct fb_info *info);
    int (*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info
    *info);
    void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect
```

```
    *rect);
    void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea
    *region);
    void (*fb_imageblit) (struct fb_info *info, const struct fb_image
    *image);
    int (*fb_cursor) (struct fb_info *info, struct fb_cursor *cursor);
    void (*fb_rotate)(struct fb_info *info, int angle);
    int (*fb_sync)(struct fb_info *info);
    int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,
            unsigned long arg);
    int (*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,
            unsigned long arg);
    int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);
    void (*fb_save_state)(struct fb_info *info);
    void (*fb_restore_state)(struct fb_info *info);
    void (*fb_get_caps)(struct fb_info *info, struct fb_blit_caps *caps,
                struct fb_var_screeninfo *var);
};
```

这个结构体内有很多函数，在这里不一一讲述了，驱动不用每个函数都去实现它，只要实现其中部分函数就可以了，比较多的是实现设备属性的设置和获得函数如 fb_get_var、fb_set_var。

fbmem.c 还实现了如下函数：

```
register_framebuffer(struct fb_info *fb_info);
unregister_framebuffer(struct fb_info *fb_info);
```

这两个是提供给下层 FrameBuffer 设备驱动的接口，设备驱动程序通过这两个函数向系统注册或注销自己。可以说底层设备驱动所要做的所有事情就是填充 fb_info 结构并向系统注册或注销它。

# 6.4　Linux 2.6.25 的 LCD 驱动源码分析

内核源码中已经有 LCD 的相驱动了，驱动开发人员只要了解了内核的 LCD 驱动体系结构，然后参考内核中已有 LCD 驱动源码，再针对具体的显示屏型号和硬件资源做相关修改就可以了。因此，从本节开始将要分析 Linux 2.6.25 的 LCD 驱动源码。

## 6.4.1　LCD 驱动开发的主要工作

LCD 驱动开发的工作包括两个方面，分别是初始化函数的编写和填充 fb_info 结构体中的主要成员函数，下面分别讲述。

### 1．编写初始化函数

初始化函数首先初始化各个 LCD 控制器寄存器，通过写寄存器来设置显示的模式和颜色数，然后在内存中分配 LCD 显示缓冲区。在 Linux 中可以用 kmalloc()函数分配一段连续的空间。缓冲区大小为：点阵行数×点阵列数×用于表示一个像素的比特数/8。缓冲

区通常分配在大容量的片外 SDRAM 中，起始地址保存在 LCD 控制寄存器中。本节采用的 LCD 显示方式为 320×240、16 位彩色，则需要分配的显示缓冲区为 320×240×2=15kb。最后是初始化一个 fb_info 结构体，填充其中的成员变量，并调用 register_framebuffer (&fb_info)，将 fb_info 注册入内核。

### 2．编写成员函数

对于嵌入式系统的简单实现，编写结构 fb_info 中函数指针 fb_ops 对应的成员函数只需要下列 3 个函数就可以了。

```
struct fb_ops{
...
int (*fb_get_fix)(struct fb_fix_screeninfo *fix, int con, struct fb_info
*info);
int (*fb_get_var)(struct fb_var_screeninfo *var, int con, struct fb_info
*info);
int (*fb_set_var)(struct fb_var_screeninfo *var, int con, struct fb_info
*info);
...
}
```

Struct fb_ops 在 include/linux/fb.h 中定义，前面已经讲述过了。这些函数都是用来设置/获取 fb_info 结构中的成员变量的。当应用程序对设备文件进行 ioctl 系统调用时会调用它们。对于 fb_get_fix()，应用程序把 fb_fix_screeninfo 这个结构传进来，在函数中对各个成员变量赋值，主要是 smem_start（缓冲区起始地址）和 smem_len（缓冲区长度），最终返回给应用程序。而 fb_set_var()函数的传入参数是 fb_var_screeninfo，函数中需要对 xres、yres 和 bits_per_pixel 赋值。

对于/dev/fb，对显示设备的操作主要有以下几种。

- 读/写（read/write）/dev/fb：相当于读/写屏幕缓冲区。
- 映射（map）操作：由于 Linux 工作在保护模式，每个应用程序都有自己的虚拟地址空间，在应用程序中是不能直接访问物理缓冲区地址的。为此，Linux 在文件操作 file_operations 结构中提供了 mmap()函数，可将文件的内容映射到用户空间。帧缓冲设备则可通过映射操作，可将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址之内，之后用户就可以通过读写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。
- I/O 控制：帧缓冲设备对设备文件的 ioctl 操作可读取/设置显示设备及屏幕的参数，如分辨率、显示颜色数和屏幕大小等。ioctl 的操作是由底层的硬件驱动程序来完成的。在应用程序中，操作/dev/fb 的一般步骤是：打开/dev/fb 设备文件；用 ioctrl 操作取得当前显示屏幕的参数，如屏幕分辨率以及每个像素的比特数，根据屏幕参数可计算屏幕缓冲区的大小；将屏幕缓冲区映射到用户空间；映射后即可直接读写屏幕缓冲区，进行绘图和图片显示了。

## 6.4.2　s3c2410fb_init()函数分析

首先要分析的是 s3c2410fb_init()函数，该函数内容相对简单，在 s3c2410fb_init()函数

体内只是包装了对平台驱动注册函数 platform_driver_register()的调用，它的参数是&s3c2410fb_driver。这里是向内核注册一个 platform 设备驱动的意思，该 platform 设备是 LCD 设备。

platform 有两个比较重要的数据结构，分别是 platform_device 和 platform_driver，这里用到的就是 platform_driver。platform_driver 在 include/linux/platform_device.h 中有定义，它的成员无非是一些回调函数还有一个同 platform_device 一致的设备名字，在前面章节已经讲过了。在 LCD 驱动程序（drivers/video/s3c2410fb.c ）中有 platform_driver 结构体的定义，内容如下：

```
static struct platform_driver s3c2410fb_driver = {
    .probe       = s3c2410fb_probe,      //初始化函数
    .remove      = s3c2410fb_remove,
    .suspend         = s3c2410fb_suspend,
    .resume      = s3c2410fb resume,
    .driver      = {
        .name    = "s3c2410-lcd",/*设备名字，这个名字一定要和struct  platform_
                                device 中的 name 域一样才能把平台驱动和前面定义的
                                平台数据联系起来*/

        .owner   = THIS MODULE,
    },
};
```

由以上可以看到该 platform 设备的驱动有 s3c2410fb_probe 、s3c2410fb_remove 等回调函数。在通过 platform_driver_register 函数注册该设备的过程中，它会回调 probe 探测函数，也就是说 s3c2410fb_probe 是在 platform_driver_registe 中被回调的。

### 6.4.3　s3c2410fb_probe()函数分析

这个函数是驱动的关键函数，下面就一步步来分析它。因为函数太长，这里先列出源码，然后在后面再对源码进行分析。

```
799  static int __init s3c24xxfb_probe(struct platform_device *pdev,
800                                  enum s3c drv type drv type)
801    {
802          struct s3c2410fb_info *info;
803          struct s3c2410fb_display *display;
804          struct fb_info *fbinfo;
805          struct s3c2410fb_mach_info *mach_info;
806          struct resource *res;
807          int ret;
808          int irq;
809          int i;
810          int size;
811          u32 lcdcon1;
812
```

第 802～811 行定义了本函数用到的变量，其中 s3c2410fb_info 是驱动自己定义的设备数据结构，可以说该结构记录了 s3c2410fb 驱动的所有信息；s3c2410fb_display 这个结构体定义了 LCD 屏的规格和时序；fb_info 是 framebuffe 里定义的设备数据结构，表示一个

显示设备，s3c2410fb_probe 的最终目的填充该结构，并向内核注册；s3c2410fb_mach_info 结构里包含了一些 LCD 控制器的 GPIO，这个主要用来设置引脚的功能。Resource 指向设备用到的 I/O 资源，这是平台驱动的结构；irq 用来保存中断号。

```
813         mach_info = pdev->dev.platform_data;
814         if (mach_info == NULL) {
815               dev err(&pdev->dev,
816                     "no platform data for lcd, cannot attach\n");
817               return -EINVAL;
818         }
819
```

第 813～818 行，用于获得平台数据，这里需要深入说明一下。mach_info 是一个 s3c2410fb_mach_info 类型的指针，它描述了 LCD 的一些属性。值得读者注意的是 s3c2410fb_mach_info 和 s3c2410fb_info 结构是有区别的。同样是描述 LCD 的属性，s3c2410fb_mach_info 只是用于描述 LCD 初始化时所用的值，而 s3c2410fb_info 是描述整个 LCD 驱动的结构体。

s3c2410fb_mach_info 在 include/asm-arm/arch-s3c2410/fb.h 中定义，它不是内核所认知的数据结构，它只和平台相关，也就是说，这只是驱动程序设计者设计的结构。从后面的 if 语句可以知道，如果 mach_info 等于 NULL，整个驱动程序就会退出，这里读者应该会问，那 pdev->dev.platform_data 是在什么时候被初始化的呢？这个问题在后面会讲到。

```
820         if (mach info->default display >= mach info->num displays) {
821            dev_err(&pdev->dev, "default is %d but only %d displays\n",
822               mach info->default display, mach info->num displays);
823               return -EINVAL;
824         }
825
826         display = mach_info->displays + mach_info->default_display;
```

第 820～826 行用来查找当前显示屏的规格。这个规格是在 arch/arm/mach-smdk2440.c 文件中定义的平台数据结构体的一些参数。

```
827
828         irq = platform get irq(pdev, 0);
829         if (irq < 0) {
830               dev_err(&pdev->dev, "no irq for device\n");
831               return -ENOENT;
832         }
```

第 829～832 行获得设备的中断号。这个中断号也是在 arch/arm/mach-smdk2440.c 文件中定义的硬件所使用到的中断资源。

```
833
834         fbinfo = framebuffer alloc(sizeof(struct s3c2410fb info),
            &pdev->dev);
835         if (!fbinfo)
836               return -ENOMEM;
```

第 834～836 行的功能是向内核申请一段大小为 sizeof(struct fb_info) + size 的空间，其中 size 的大小代表设备的私有数据空间，并用 fb_info 的 par 域指向该私有空间。

```
837
838         platform_set_drvdata(pdev, fbinfo);
```

```
839
840         info = fbinfo->par;
841         info->dev = &pdev->dev;
842         info->drv_type = drv_type;
```

第 838～842 行设置驱动数据并填充驱动数据。

```
843
844         res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
845         if (res == NULL) {
846                 dev_err(&pdev->dev, "failed to get memory registers\n");
847                 ret = -ENXIO;
848                 goto dealloc_fb;
849         }
850
851         size = (res->end - res->start) + 1;
852         info->mem = request_mem_region(res->start, size, pdev->name);
853         if (info->mem == NULL) {
854                 dev_err(&pdev->dev, "failed to get memory region\n");
855                 ret = -ENOENT;
856                 goto dealloc_fb;
857         }
858
859         info->io = ioremap(res->start, size);
860         if (info->io == NULL) {
861                 dev_err(&pdev->dev, "ioremap() of registers failed\n");
862                 ret = -ENXIO;
863                 goto release_mem;
864         }
865
866         info->irq_base = info->io + ((drv_type == DRV_S3C2412) ?
            S3C2412_LCDINTBASE : S3C2410_LCDINTBASE);
```

第 844～866 行，主要是映射内核资源，如 I/O 口和中断号，平台数据传递不定期来的硬件资源地址都是物理地址，而在内核中对硬件进行访问都是要通过虚拟地址的，所以要用相关函数把物理地址映射为虚拟地址。如果在这里映射失败将退出程序。

```
867
868         dprintk("devinit\n");
869
870         strcpy(fbinfo->fix.id, driver_name);
871
872         /*Stop the video*/
873         lcdcon1 = readl(info->io + S3C2410_LCDCON1);
874         writel(lcdcon1 & ~S3C2410_LCDCON1_ENVID, info->io + S3C2410_
            LCDCON1);
875
876         fbinfo->fix.type            = FB_TYPE_PACKED_PIXELS;
877         fbinfo->fix.type_aux        = 0;
878         fbinfo->fix.xpanstep        = 0;
879         fbinfo->fix.ypanstep        = 0;
880         fbinfo->fix.ywrapstep       = 0;
881         fbinfo->fix.accel           = FB_ACCEL_NONE;
882
883         fbinfo->var.nonstd          = 0;
884         fbinfo->var.activate        = FB_ACTIVATE_NOW;
885         fbinfo->var.accel_flags     = 0;
886         fbinfo->var.vmode           = FB_VMODE_NONINTERLACED;
887
888         fbinfo->fbops               = &s3c2410fb_ops;
```

```
889         fbinfo->flags               = FBINFO_FLAG_DEFAULT;
890         fbinfo->pseudo_palette      = &info->pseudo_pal;
891
892         for (i = 0; i < 256; i++)
893               info->palette_buffer[i] = PALETTE_BUFF_CLEAR;
```

第 872～893 行首先关闭显示屏，然后对 fbinfo 进行附值。

```
894
895         ret = request_irq(irq, s3c2410fb_irq, IRQF_DISABLED, pdev->
            name, info);
896         if (ret) {
897            dev_err(&pdev->dev, "cannot get irq %d - err %d\n", irq, ret);
898               ret = -EBUSY;
899               goto release regs;
900         }
901
902         info->clk = clk get(NULL, "lcd");
903         if (!info->clk || IS_ERR(info->clk)) {
904               printk(KERN_ERR "failed to get lcd clock source\n");
905               ret = -ENOENT;
906               goto release_irq;
907         }
908
909         clk_enable(info->clk);
910         dprintk("got and enabled clock\n");
911
912         msleep(1);
```

第 895～912 行申请中断并打开 LCD 时钟，使得各 LCD 相关引脚输出信号。

```
913
914         /*find maximum required memory size for display*/
915         for (i = 0; i < mach info->num displays; i++) {
916               unsigned long smem_len = mach_info->displays[i].xres;
917
918               smem_len *= mach_info->displays[i].yres;
919               smem_len *= mach_info->displays[i].bpp;
920               smem len >>= 3;
921               if (fbinfo->fix.smem_len < smem_len)
922                     fbinfo->fix.smem_len = smem_len;
923         }
924
925         /*Initialize video memory*/
926         ret = s3c2410fb map video memory(fbinfo);
927         if (ret) {
928             printk(KERN ERR "Failed to allocate video RAM: %d\n", ret);
929               ret = -ENOMEM;
930               goto release_clock;
931         }
```

第 915～931 行计算显示内存容量并映射 DMA 资源，上层应用程序把要显示的图像存入这个内存区域，然后 LCD 控制器通过 DMA 从这个映射内容中取出要显示的图像。

```
932
933         dprintk("got video memory\n");
934
935         fbinfo->var.xres = display->xres;
936         fbinfo->var.yres = display->yres;
937         fbinfo->var.bits per pixel = display->bpp;
938
```

```
939         s3c2410fb_init_registers(fbinfo);
940
941         s3c2410fb check var(&fbinfo->var, fbinfo);
942
943         ret = register_framebuffer(fbinfo);
944         if (ret < 0) {
945           printk(KERN_ERR "Failed to register framebuffer device: %d\n",
946                     ret);
947                 goto free video memory;
948         }
949
950         /*create device files*/
951         device_create_file(&pdev->dev, &dev_attr_debug);
952
953         printk(KERN_INFO "fb%d: %s frame buffer device\n",
```

第 939～953 行，初始化各个 LCD 控制寄存器、注册我们的 fbinfo 并为该设备创建一个在 sysfs 中的属性。

```
954                 fbinfo->node, fbinfo->fix.id);
955
956         return 0;
957
958 free video memory:
959         s3c2410fb_unmap_video_memory(fbinfo);
960 release_clock:
961         clk disable(info->clk);
962         clk_put(info->clk);
963 release_irq:
964         free irq(irq, info);
965 release_regs:
966         iounmap(info->io);
967 release mem:
968         release_resource(info->mem);
969         kfree(info->mem);
970 dealloc fb:
971         platform_set_drvdata(pdev, NULL);
972         framebuffer_release(fbinfo);
973         return ret;
974 }
```

前面留下了一个问题，pdev->dev.platform_data 是在什么时候被初始化的？其实在内核启动 init 进程之前就会执行 smdk2410_map_io()函数，而在 smdk2410_map_io()函数中加入了 s3c24xx_fb_set_platdata (&smdk2410_lcd_platdata)这条语句，s3c24xx_fb_set_platdata()的实现为：

```
void __init s3c24xx_fb_set_platdata(struct s3c2410fb_mach_info *pd)
{
    s3c_device_lcd.dev.platform_data = pd;
}
```

根据这些代码，可以清楚地看到 s3c_device_lcd.dev.platform_data 指向了 smdk2410_lcd_platdata，而这个 smdk2410_lcd_platdata 就是一个 s3c2410fb_mach_info 的变量，它里面就存放了 LCD 驱动初始化需要的初始数据。当 s3c2410fb_probe 被回调时，所传给它的参数实际就是 s3c_device_lcd 的首地址。

在 s3c2410fb_probe 中最后调用了 s3c2410fb_init_registers()和 s3c2410fb_check_var()函数，这里应该将它们交代清楚。很明显，s3c2410fb_init_registers()是用来初始化相关寄存

器的函数。那么后者呢？这里先说 s3c2410fb_init_registers()。s3c2410fb_init_registers()的定义和实现如下，先根据它的程序流程，一步一步分析。

```
static int s3c2410fb_init_registers(struct s3c2410fb_info *fbi)
{
      unsigned long flags;
      /*Initialise LCD with values from haret*/
      local_irq_save(flags);    /*关闭中断，在关闭中断前，中断的当前状态被保存在
                                flags 中，对于关闭中断的函数，Linux 内核有很多种，
                                可以查阅相关的资料。*/

/*下面的 modify_gpio()函数是修改处理器 GPIO 的工作模式，它的实现很简单，将第 2 个参数
的值与第 3 个参数的反码按位与操作后，写到第 1 个参数里。这里的第 1 个参数实际就是硬件的
GPIO 控制器。*/
      modify_gpio(S3C2410_GPCUP, mach_info->gpcup,  mach_info->gpcup_mask);
      modify_gpio(S3C2410_GPCCON, mach_info->gpccon, mach_info->gpccon_
      mask);
      modify_gpio(S3C2410_GPDUP, mach_info->gpdup,  mach_info->gpdup_mask);
      modify_gpio(S3C2410_GPDCON, mach_info->gpdcon, mach_info->gpdcon_
      mask);
      local_irq_restore(flags);    /*使能中断，并恢复以前的状态*/
  /*下面的几个 writel()函数开始初始化 LCD 控制寄存器，它的值就是在 smdk2410_lcd_
  platdata(arch/arm/mach-s3c2410/mach-smdk2410.c)中 regs 域的值。*/
      writel(fbi->regs.lcdcon1, S3C2410_LCDCON1);
      writel(fbi->regs.lcdcon2, S3C2410_LCDCON2);
      writel(fbi->regs.lcdcon3, S3C2410_LCDCON3);
      writel(fbi->regs.lcdcon4, S3C2410_LCDCON4);
      writel(fbi->regs.lcdcon5, S3C2410_LCDCON5);
     s3c2410fb_set_lcdaddr(fbi);    /*该函数的主要作用是让处理器的 LCD 控制器的 3
                                    个地址寄存器指向正确的位置，这个位置就是 LCD
                                    的缓冲区，详细的情况可以参见 s3c2410 的用户手
                                    册。*/
      /*下面的程序是打开 video，在 s3c2410fb_probe 中被关闭了，这里打开*/
      fbi->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID;
      writel(fbi->regs.lcdcon1, S3C2410_LCDCON1);
      return 0;
}
```

s3c2410fb_init_registers 就简单分析到这里。下面看看 s3c2410fb_check_var()函数要做些什么事，要说到这个函数，还得提到 fb_var_screeninfo 这个结构类型，与它对应的是 fb_fix_screeninfo 结构类型。这两个类型分别代表了显示屏的属性信息，这些信息可以分为可变属性信息（如颜色深度、分辨率等）和不可变的信息（如帧缓冲的真实地址）。既然 fb_var_screeninfo 表示了可变的属性信息，那么这些可变的信息就应该有一定的范围，否则显示就会出问题，所以 s3c2410fb_check_var()函数的功能就是要在 LCD 的帧缓冲驱动开始运行之前将这些值初始到合法的范围内。知道了 s3c2410fb_check_var()函数要做什么，再去阅读 s3c2410fb_check_var()函数的代码就没什么问题了。

## 6.4.4　s3c2410fb_remove()函数分析

现在解释 s3c2410fb_driver 中的最后一个关键函数 s3c2410fb_remove()。顾名思义该函

数要将这个 platform 设备从系统中移除，可以推测它的作用应该释放掉所有的资源，包括内存空间。中断线等。和前文一样，我们在它的实现代码中一步步解释。

```
static int s3c2410fb_remove(struct platform_device *pdev)
{
      struct fb_info     *fbinfo = platform_get_drvdata(pdev);
                                  /*该函数从 platform_device 中获得 fb_info 信息*/
      struct s3c2410fb_info *info = fbinfo->par;   //得到私有数据
      int irq;
      s3c2410fb_stop_lcd(info);                    //该函数停止 LCD 控制器，
                                                   实现可以在 s3c2410fb.c 中
                                                   找到
      msleep(1); //等待 LCD 停止
      s3c2410fb_unmap_video_memory(info);          //该函数释放缓冲区
     if (info->clk) {                              //停止时钟
           clk disable(info->clk);
           clk_put(info->clk);
           info->clk = NULL;
      }
      irq = platform_get_irq(pdev, 0);             //得到中断线，以便释放
      free_irq(irq,info);                          //释放该中断
      release_mem_region((unsigned long)S3C24XX_VA_LCD, S3C24XX_SZ_LCD);
                                                   //释放内存空间
      unregister_framebuffer(fbinfo);              //向内核注销该帧缓冲
      return 0;
}
```

# 6.5　移植内核中的 LCD 驱动

不同的开发者在设计同一种设备时都有其独特性，比如人、访问硬件资源所对应的端口地址等。所以驱动移植的首要工作是要明确所驱动的设备硬件连接情况，然后在此基础上对驱动源码进行修改以适应具体的硬件。

## 6.5.1　LCD 硬件电路图

写驱动程序前先了解一下电路连接情况。LCD 的像素同步时钟信号、水平同步信号、垂直同步信号直接连接到 LCD 的 VCLK、VLINE 和 VFRAME 上，用 GPG4 作为 LCD 的电源信号直接连接到 LCD_PWREN 上。如图 6.7 为 TFT LCD 屏与 CPU 的连接图。

## 6.5.2　修改 LCD 源码

本章用的源码是 linux2.6.25.8 中的 LCD 驱动源码，文件为 drivers/video/s3c2410fb.c 和 drivers/video/s3c2410fb.h。

（1）修改 arch/asm-arm/arch-s3c2410/fb.h 中的 s3c2410fb_display 结构，在其中加入一个元素用于在初始化时设置 CLKVAL 参数。修改后结构如下，黑体为修改部分。

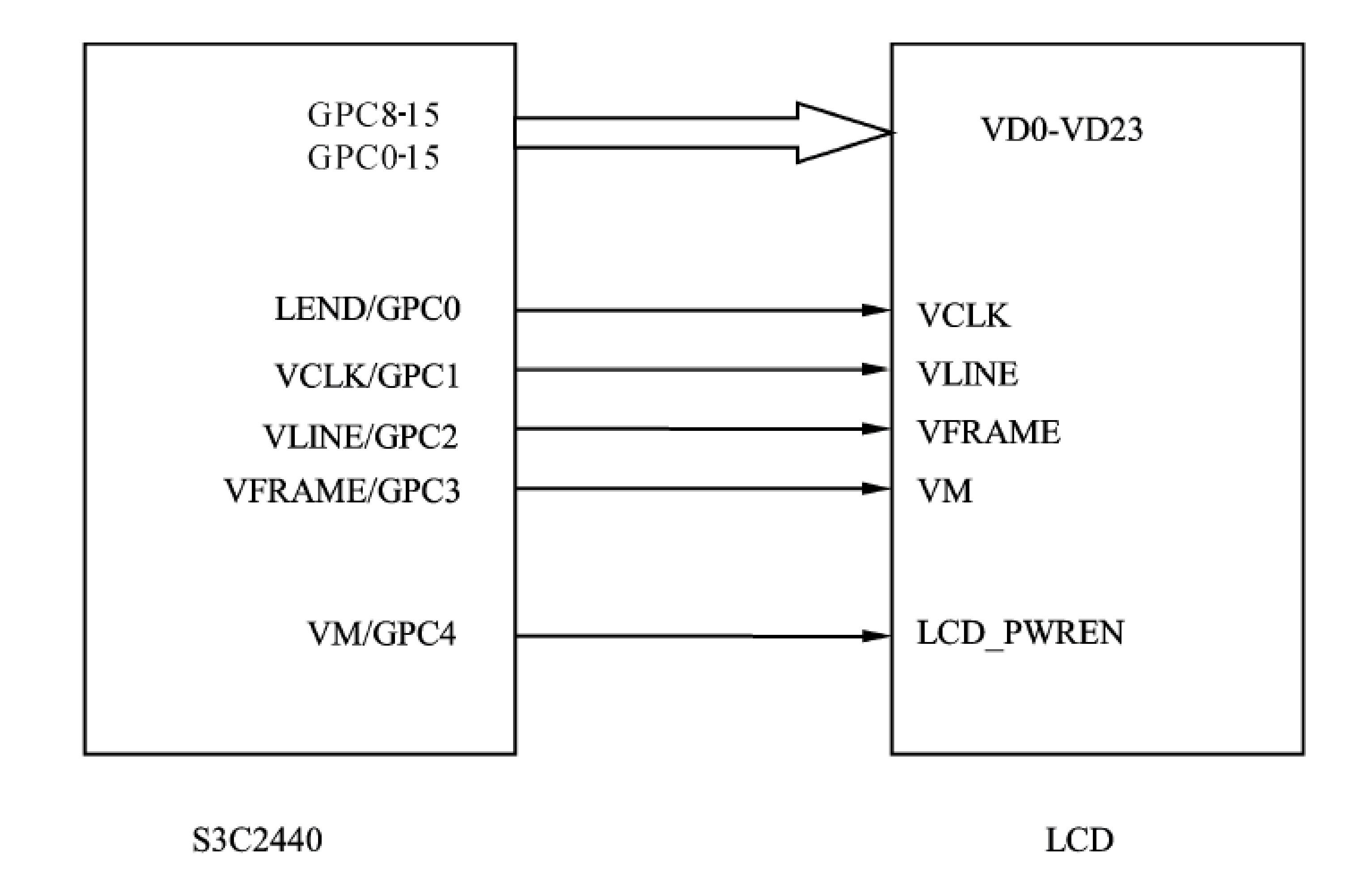



图 6.7　TFT LCD 屏与 CPU 的连接图

```
struct s3c2410fb display {
    /*LCD type*/
    unsigned type;

    /*Screen size*/
    unsigned short width;
    unsigned short height;

    /*Screen info*/
    unsigned short xres;
    unsigned short yres;
    unsigned short bpp;

    unsigned pixclock;      //pixclock in picoseconds
    unsigned setclkval;     //在这里加入 setclkval ，用来设置 CLKVAL 参数
    unsigned short left_margin;     //(TFT)或 HCLKs(STN)的像素值
    unsigned short right_margin;    //(TFT)或 HCLKs(STN)的像素值
    unsigned short hsync_len;       //(TFT)或 HCLKs(STN)的像素值
    unsigned short upper_margin;    //(TFT)或 0(STN)的行
    unsigned short lower_margin;    //(TFT)或 0(STN)的行
    unsigned short vsync_len;       //(TFT)或 0(STN)的行

    // lcd configuration registers
    unsigned long  lcdcon5;
};
```

（2）在源码中加入设置 CLKVAL 参数的代码，这里修改 drivers/video/s3c2410fb.c 中的函数 s3c2410fb_activate_var()，修改后结果如下，黑体为修改部分。

```
static void s3c2410fb_activate_var(struct fb_info *info)
{
    struct s3c2410fb_info *fbi = info->par;
    void __iomem *regs = fbi->io;
    int type = fbi->regs.lcdcon1 & S3C2410_LCDCON1_TFT;
```

```
    struct fb_var_screeninfo *var = &info->var;
    int clkdiv = s3c2410fb calc pixclk(fbi, var->pixclock) / 2;

    /*获取平台数据并得到参数值*/
struct s3c2410fb_mach_info *mach_info=fbi->dev->platform_data;
struct s3c2410fb_display *default_display=mach_info->displays+mach_info->
default_display;

    dprintk("%s: var->xres  = %d\n",   FUNCTION  , var->xres);
    dprintk("%s: var->yres  = %d\n",   FUNCTION  , var->yres);
    dprintk("%s: var->bpp   = %d\n", __FUNCTION__, var->bits_per_pixel);

    if (type == S3C2410_LCDCON1_TFT) {
        s3c2410fb_calculate_tft_lcd_regs(info, &fbi->regs);
        --clkdiv;
        if (clkdiv < 0)
            clkdiv = 0;
    } else {
        s3c2410fb calculate stn lcd regs(info, &fbi->regs);
        if (clkdiv < 2)
            clkdiv = 2;
    }

    fbi->regs.lcdcon1 |=  S3C2410_LCDCON1_CLKVAL(clkdiv);

    /*写新的寄存器*/

    dprintk("new register set:\n");
    dprintk("lcdcon[1] = 0x%08lx\n", fbi->regs.lcdcon1);
    dprintk("lcdcon[2] = 0x%08lx\n", fbi->regs.lcdcon2);
    dprintk("lcdcon[3] = 0x%08lx\n", fbi->regs.lcdcon3);
    dprintk("lcdcon[4] = 0x%08lx\n", fbi->regs.lcdcon4);
    dprintk("lcdcon[5] = 0x%08lx\n", fbi->regs.lcdcon5);

    writel(fbi->regs.lcdcon1 & ~S3C2410_LCDCON1_ENVID,
        regs + S3C2410_LCDCON1);
    writel(fbi->regs.lcdcon2, regs + S3C2410 LCDCON2);
    writel(fbi->regs.lcdcon3, regs + S3C2410_LCDCON3);
    writel(fbi->regs.lcdcon4, regs + S3C2410 LCDCON4);
    writel(fbi->regs.lcdcon5, regs + S3C2410_LCDCON5);

    /*set lcd address pointers*/
    s3c2410fb_set_lcdaddr(info);

    /*注释源码中设置 CLKVAL 的代码并增加我们的设置代码*/
    /*fbi->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID,*/
fbi->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID (default_display->setclkval) ;

    writel(fbi->regs.lcdcon1, regs + S3C2410_LCDCON1);
}
```

（3）修改 LCD 的参数值。源码中的参数设置在 arch/arm/mach-smdk2440.c 中，全局变量 smdk2440_lcd_cfg 就是配置 LCD 参数的地方，配置这些参数要根据具体 LCD 屏手册来配置，读者请参考自己所使用的 LCD 屏的 datasheet 来设置。本书使用的是型号为 WXCAT35-TG3#001F 的 LCD 屏，其参数如表 6.16 所示。

表 6.16　LCD屏的时序表

| Signal | Item | Symbol | Min | Typ | Max | Unit |
|---|---|---|---|---|---|---|
| Dclk | Frequency | Dclk | - | 6.4 | - | MHZ |
| | Dclk-Period | Tosc | - | 156 | - | ns |
| Data | Setup Time | TSU | 12 | - | - | ns |
| | Hold Time | THD | 12 | - | - | ns |
| Hsync | Period | Th | - | 408 | - | DCLK |
| | Pulse Width | Thp | - | 30 | - | DCLK |
| | Back-Porch | Thb | - | 38 | - | DCLK |
| | Display Period | Thd | - | 320 | - | DCLK |
| | Front-Porch | Thf | - | 20 | - | DCLK |
| Vsync | Period | Tv | - | 270 | - | TH |
| | Pulse Width | Tvp | - | 3 | - | TH |
| | Back-Porch | Tvb | - | 15 | - | TH |
| | Display Period | Tvd | - | 240 | - | TH |
| | Front-Porch | Tvf | - | 12 | - | TH |

修改后结果如下，黑体为修改的内容。

```
static struct s3c2410fb_display smdk2440_lcd_cfg __initdata = {

    .lcdcon5     = S3C2410_LCDCON5_FRM565 |
              S3C2410_LCDCON5_INVVLINE |
              S3C2410_LCDCON5_INVVFRAME |
              S3C2410_LCDCON5_PWREN |
              S3C2410_LCDCON5_HWSWP,

    .type        = S3C2410_LCDCON1_TFT,

    .width       = 320,       //这是 LCD 屏的分辨率
    .height      = 240,

    .pixclock    = 100000,    // HCLK 60 MHz, divisor 10
    .setclkval    =0x3,       //这是前面加入的新变量
.xres        = 320,
    .yres        = 240,
    .bpp         = 16,
    /*下面写参数要对照芯片手册来修改*/
    .left_margin    = 38,     //左边筐
    .right_margin   = 20,     //右边筐
    .hsync_len  = 30,         //水平时长
    .upper_margin   = 15,     //上边筐
    .lower_margin   = 12,     //下边筐
    .vsync_len  = 3,          //垂直时长
};

static struct s3c2410fb_mach_info smdk2440_fb_info __initdata = {
    .displays    = &smdk2440_lcd_cfg,
    .num_displays   = 1,
    .default_display = 0,

#if 0
    /*currently setup by downloader*/
```

```
    .gpccon      = 0xaa940659,
    .gpccon_mask    = 0xffffffff,
    .gpcup       = 0x0000ffff,
    .gpcup_mask = 0xffffffff,
    .gpdcon      = 0xaa84aaa0,
    .gpdcon mask    = 0xffffffff,
    .gpdup       = 0x0000faff,
    .gpdup_mask = 0xffffffff,
#endif
    /*源码中把对引脚的功能设置屏蔽掉了，这样引脚的功能还是默认设置，所以要根据芯片手册
    把相关引脚设置为第三功能，即 LCD 的相关功能，设置的结果如下*/
    .gpccon      = 0xaaaaaaaa,
    .gpccon mask    = 0xffffffff,
    .gpcup       = 0xffffffff,
    .gpcup mask = 0xffffffff,
    .gpdcon      = 0xaaaaaaaa,
    .gpdcon_mask    = 0xffffffff,
    .gpdup       = 0xffffffff,
    .gpdup mask = 0xffffffff,

    //我们用的不是 LPC 屏，所以去掉这个设置
    //.lpcsel       = ((0xCE6) & ~7) | 1<<4,
};
```

（4）修改源码中一处有误的地方。在源码中，没有设置 LCD 的供电电源，这样 LCD 屏是点不起来的，所以要补充对 LCD 屏供电引脚的设置。代码如下，黑体为增加的代码。

```
/*
 * s3c2410fb_init_registers - Initialise all LCD-related registers
*/
static int s3c2410fb_init_registers(struct fb_info *info)
{
    struct s3c2410fb_info *fbi = info->par;
    struct s3c2410fb_mach_info *mach_info = fbi->dev->platform_data;
    unsigned long flags;
    void __iomem *regs = fbi->io;
    void __iomem *tpal;
    void __iomem *lpcsel;

    if (is_s3c2412(fbi)) {
        tpal = regs + S3C2412_TPAL;
        lpcsel = regs + S3C2412 TCONSEL;
    } else {
        tpal = regs + S3C2410 TPAL;
        lpcsel = regs + S3C2410 LPCSEL;
    }

    /*Initialise LCD with values from haret*/

    local irq save(flags);

    /*modify the gpio(s) with interrupts set (bjd)*/
    printk(KERN INFO "gpcup    = 0x%08lx\n", mach info->gpcup);
    printk(KERN_INFO "gpccon    = 0x%08lx\n", mach_info->gpccon);
    printk(KERN INFO "gpdup    = 0x%08lx\n", mach info->gpdup);
    printk(KERN_INFO "gpdcon    = 0x%08lx\n", mach_info->gpdcon);
    printk(KERN_INFO "gpcup_mask    = 0x%08lx\n", mach_info->gpcup_mask);
    printk(KERN INFO "gpccon mask    = 0x%08lx\n", mach info->gpccon
    mask);
```

```
    printk(KERN_INFO "gpdup_mask    = 0x%08lx\n", mach_info->gpdup_mask);
    printk(KERN_INFO "gpdcon_mask    = 0x%08lx\n", mach_info->gpdcon_
    mask);

    modify_gpio(S3C2410_GPCUP,  mach_info->gpcup,  mach_info->gpcup_mask);
    modify_gpio(S3C2410_GPCCON, mach_info->gpccon, mach_info->gpccon_mask);
    modify_gpio(S3C2410_GPDUP,  mach_info->gpdup,  mach_info->gpdup_mask);
    modify_gpio(S3C2410_GPDCON, mach_info->gpdcon, mach_info->gpdcon_mask);

   //设置 LCD 的供电电源，GPG4 为上拉，输出
    modify_gpio(S3C2410_GPGUP,  0x00000010,  0x00000010);
    modify_gpio(S3C2410_GPGCON, 0x00000300, 0x00000300);

    local_irq_restore(flags);

    printk("LPCSEL    = 0x%08lx\n", mach_info->lpcsel);
    writel(mach_info->lpcsel, lpcsel);

    printk("replacing TPAL %08x\n", readl(tpal));

    /*ensure temporary palette disabled*/
    writel(0x00, tpal);

    return 0;
}
```

## 6.5.3　配置内核

做完以上工作，我们就可以对 LCD 进行配置了。进入内核源码所在的目录，输入 make menuconfig 进入配置菜单后，进行如下配置：

（1）选择进入 Device Drivers。Linux 的所有设备驱动都放在 Device 文件夹里，而 Device 包含的所有设备驱动都会在 Device Drivers 这个选项里列出，要编译设备驱动程序都要进入这个选项然后根据具体情况进行配置，所以这里选择 Device Drivers，如图 6.8 所示。

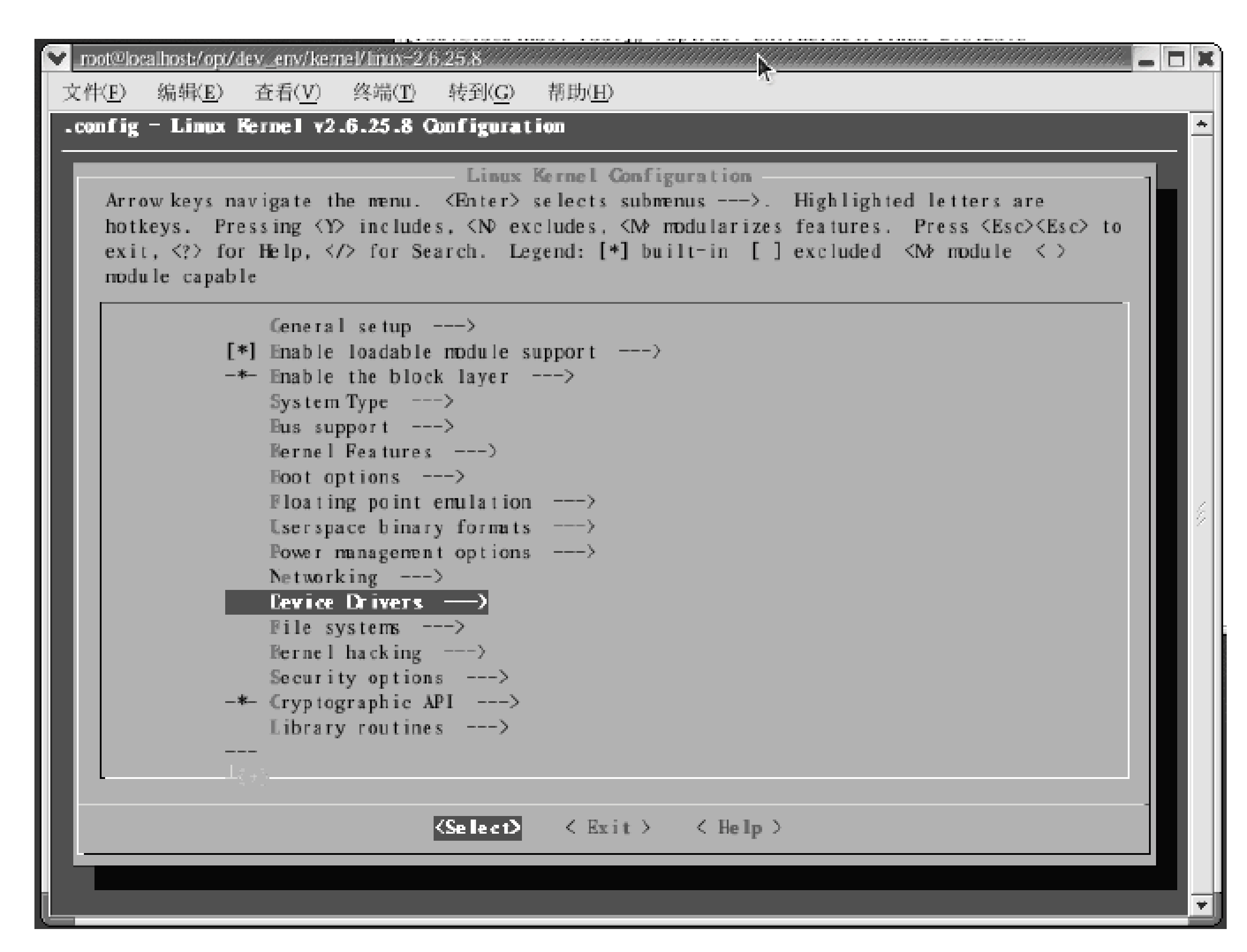



图 6.8　步骤 1

（2）选择 Graphics support 选项，如图 6.9 所示。

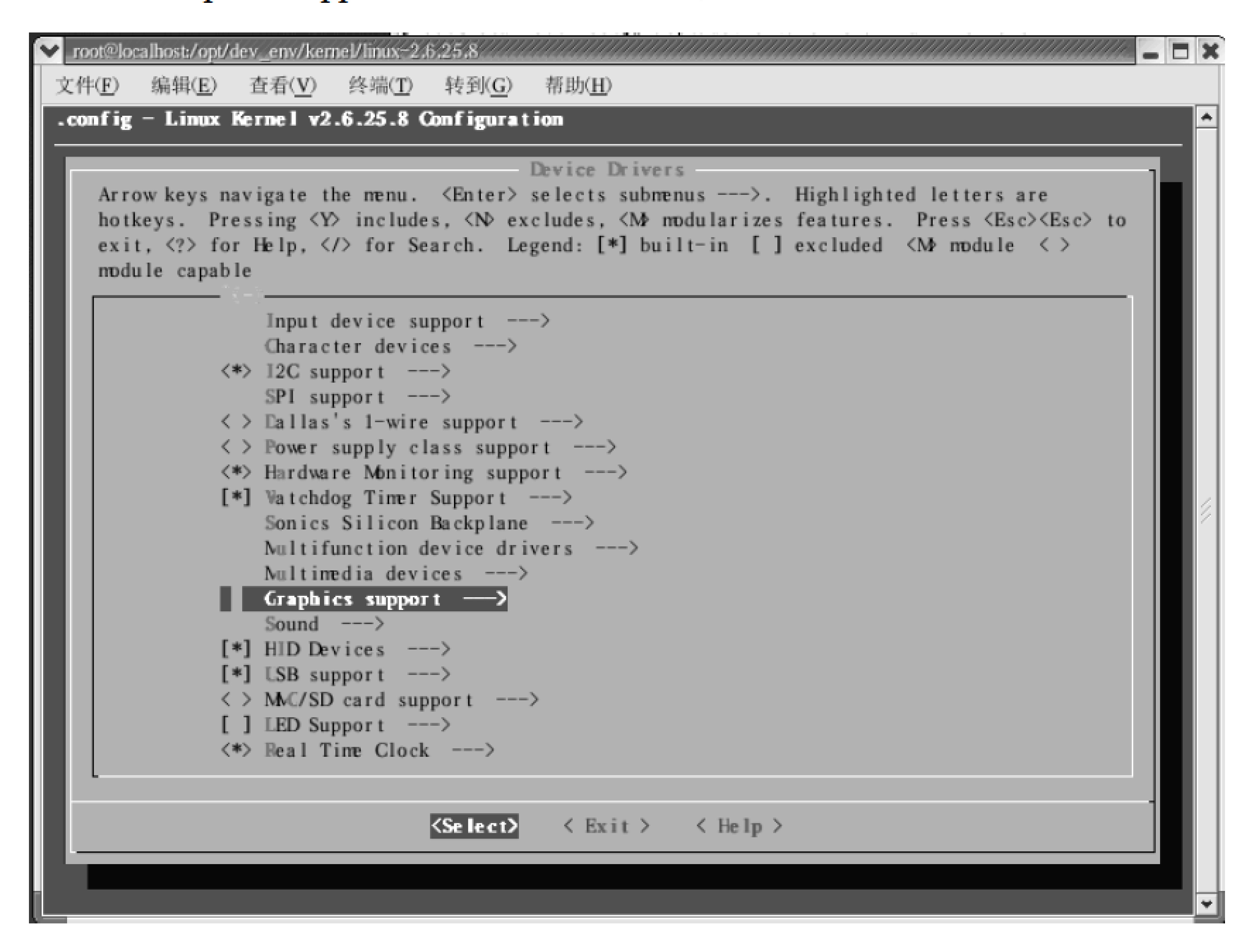



图 6.9　步骤 2

（3）选择 Support for frame buffer devices 选项，如图 6.10 所示。

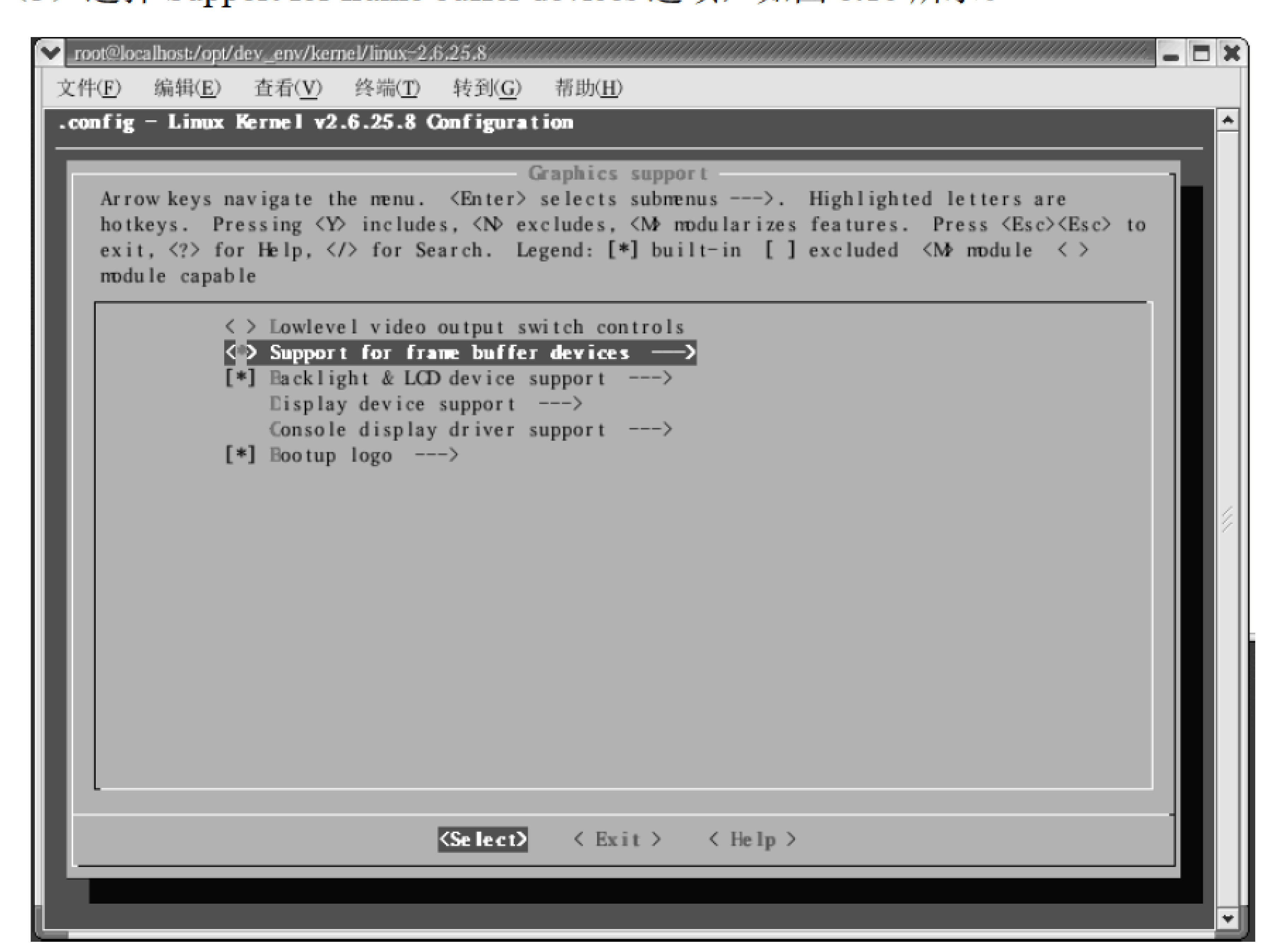



图 6.10　步骤 3

（4）在驱动程序的调试阶段最好选中 S3C2410 lcd debug messages 选项，如图 6.11 所示。这样会在标准输出中打印出调试信息，有助于驱动程序的调试。

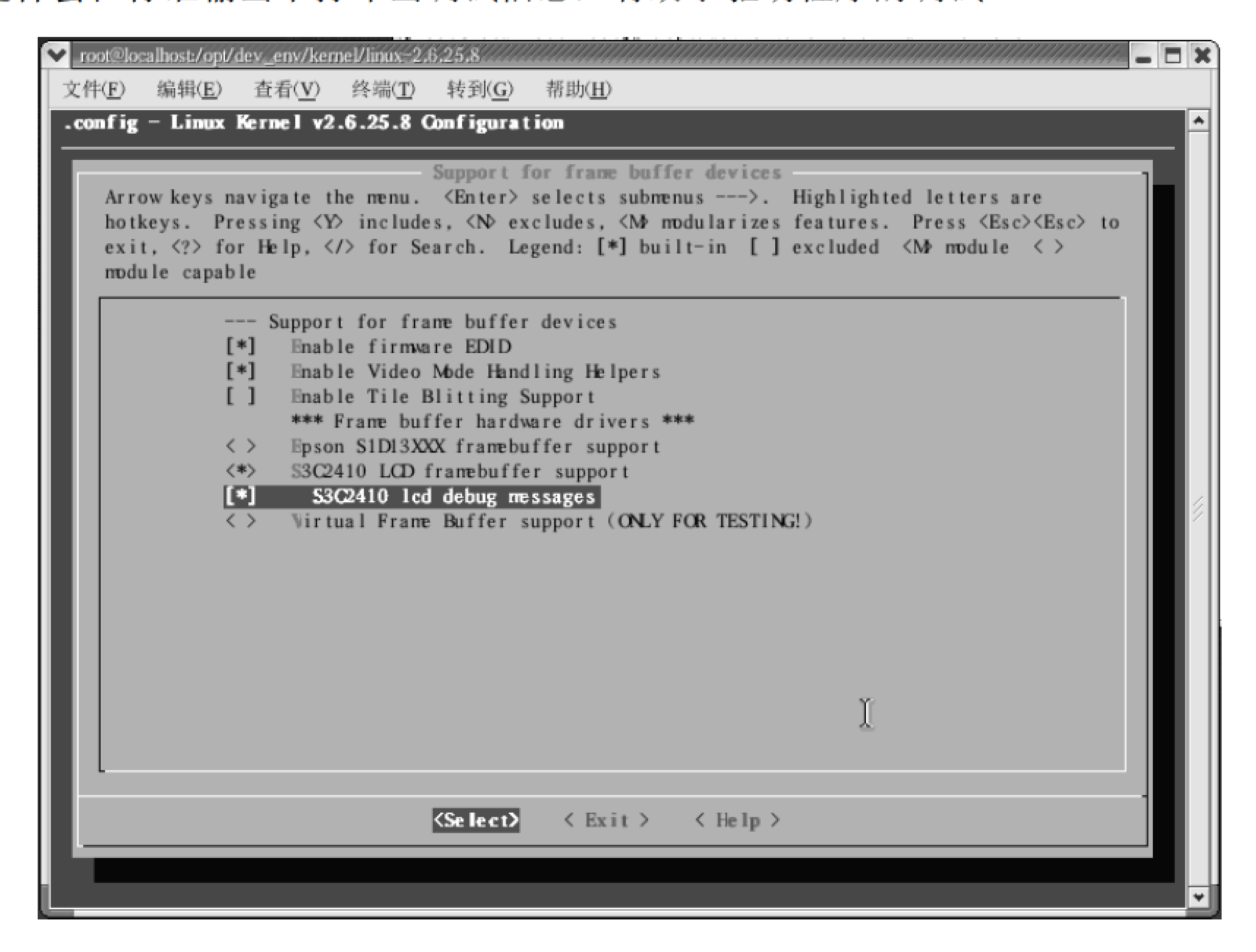



图 6.11　步骤 4

（5）增加开机 LOGO，让系统开机时在 LCD 屏上显示一个小企鹅的图片。在步骤（3）中同时选择 Bootup logo 选项，如图 6.12 所示。

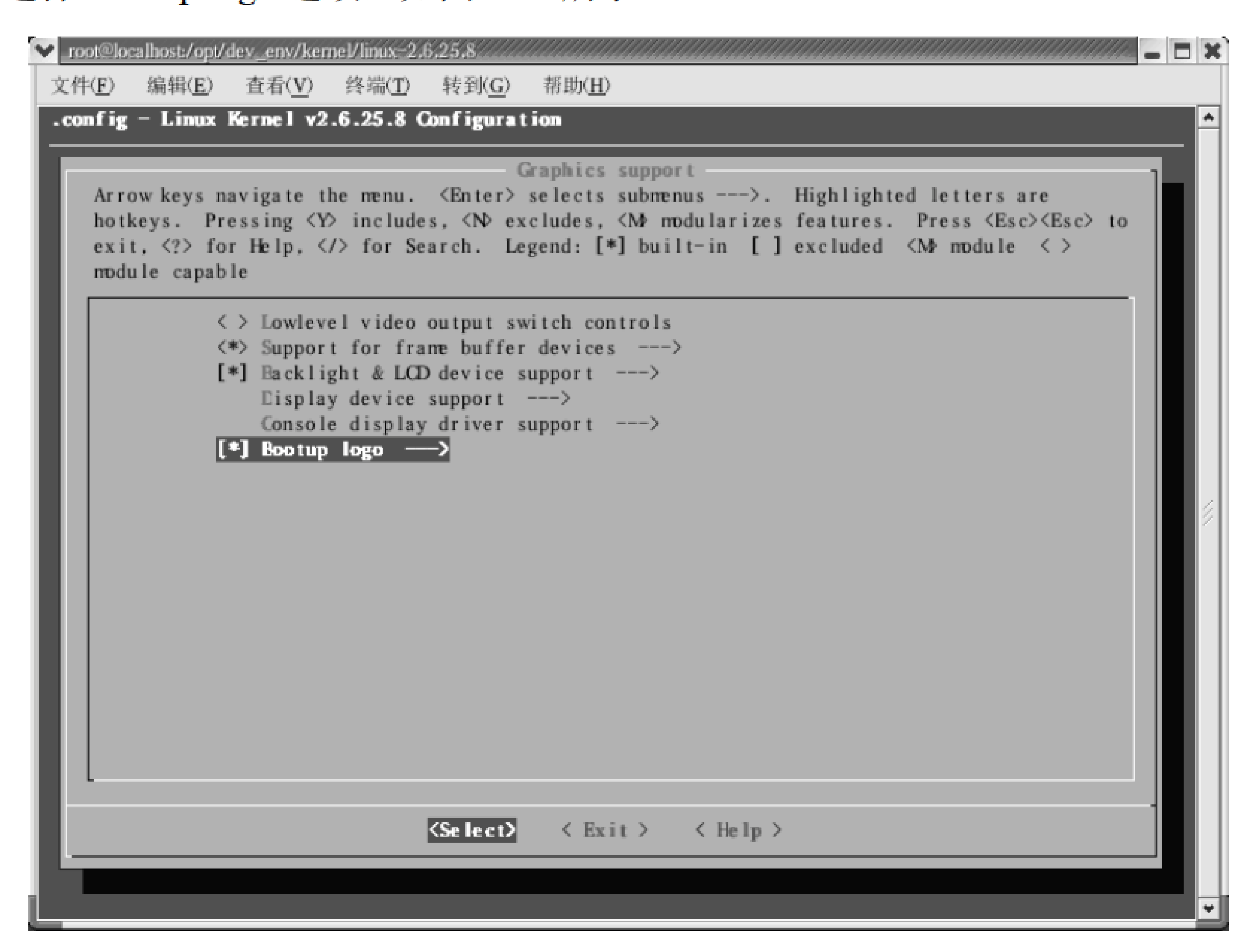



图 6.12　步骤 5

（6）进入开机 LOGO 的设置选项 Bootup logo 目录后，里面有单色的 LOGO 图片选项，也有 16 色和 224 色的 LOGO 图片选项供选择，这里根据实际情况选择 Standard 224-color Linux logo，如图 6.13 所示。这是因为我们用的 LCD 屏是支持 224 色的。

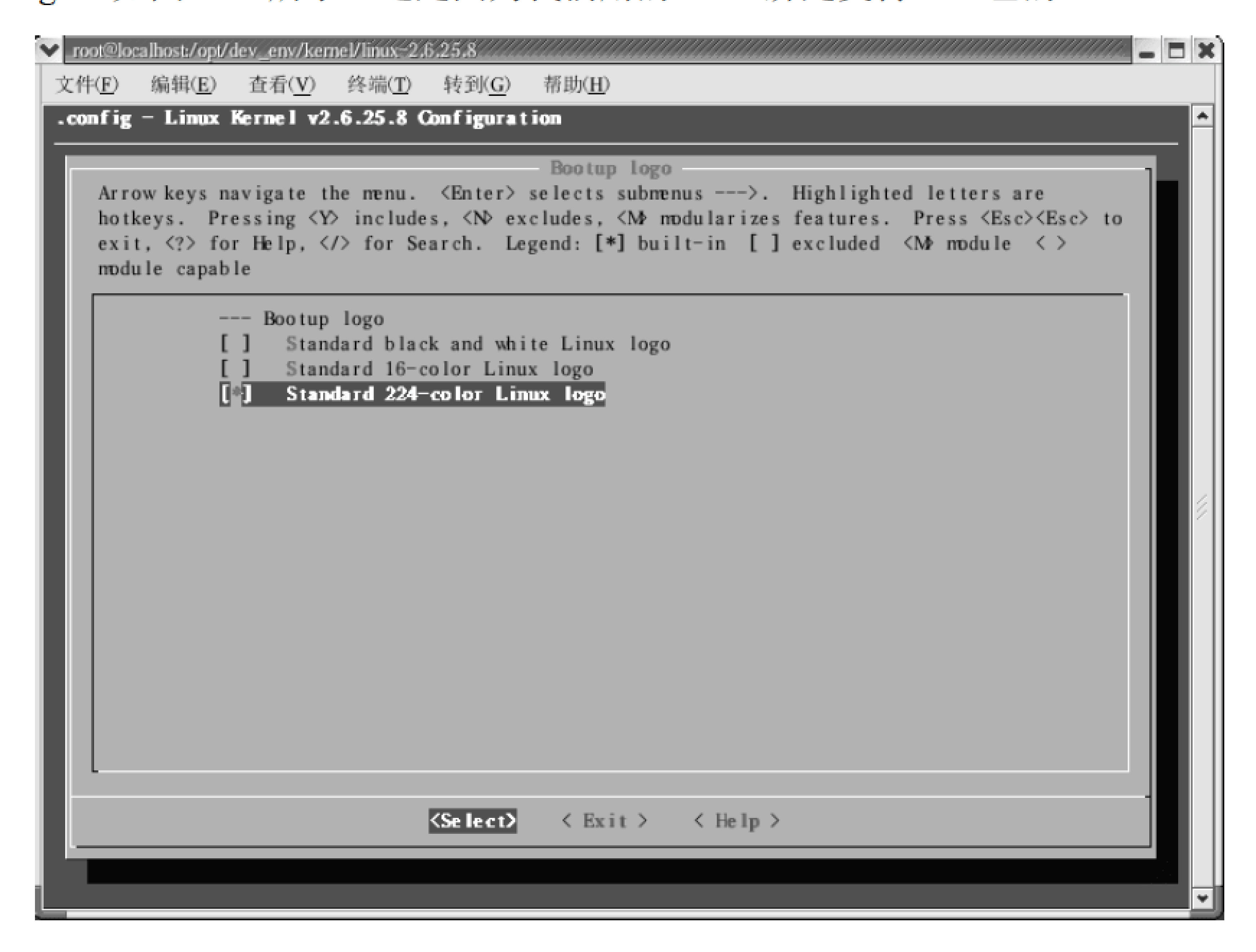



图 6.13　步骤 6

做完以上配置后保存退出配置界面，在源码目录上输入 make 重新编译内核。把内核下载到开发板上，然后开机就可以看到在 LCD 屏上有一个企鹅的图片了，说明 LCD 已经可以使用了。

## 6.6　小　　结

本章主要在分析了 S3C2440 LCD 控制寄存器的硬件操作、分析了内核中自带的 LCD 驱动源程序并在此基础上讲述了 LCD 网卡驱动的移植。LCD 驱动程序移植主要是要了解内核中 FrameBuffer 驱动程序的体系结构及如何操作 LCD 控制器，因此本章的 6.1、6.2 节是后面移植工作的基础，读者要认真掌握，特别是对 FrameBuffer 数据结构的主要成员要掌握其代表的具体含义。

# 第 7 章　触摸屏驱动移植

随着多媒体信息查询的日新月异，人们越来越多地接触触摸屏设备。触摸屏作为一种新颖的电脑输入设备，是目前最简单、方便、自然的一种人机交互方式，许多现代手持或终端设备都要用到触摸屏。触摸屏以其易于使用、坚固耐用、反应速度快、节省空间等优点，使得系统设计师们越来越多地感到使用触摸屏的确具有相当大的优越性。从本章开始将要学习 Linux 触摸屏驱动程序的移植。本章先对触摸屏做一个简单概述，然后在分析完 Linux 2.6 内核触摸屏驱动源码之后，讲述触摸屏驱动的移植，最后结合触摸屏驱动程序介绍 Linux 的内核输入子系统。

## 7.1　触摸屏概述

触控屏（Touch panel）又称为触控面板，它并不是人们日常所见的立方体屏，它只是覆盖显示屏表面的一层薄片，是一个可接收触头等输入信号的感应式液晶显示装置，它的工作原理比较简单。当接触到屏幕上的图形按钮时，屏幕上的触觉反馈系统可根据预先编程的程式驱动各种连接装置，可用以取代机械式的按钮面板，并借由液晶显示画面制造出生动的影音效果。

### 7.1.1　触摸屏工作原理

触摸屏简单地说就是一种特殊的输入设备。为了操作方便，人们用触摸屏取代鼠标或者键盘。在工作时，必须首先用手指或其他物体触摸安装在显示器前面的触摸屏，然后系统根据手指触摸的图标或菜单位置来定位选择信息输入。触摸屏由触摸检测部件和触摸屏控制器组成；触摸检测部件安装在显示器屏幕前面，用来检测用户触摸的位置，接受后送触摸屏控制器；而触摸屏控制器的主要作用是从触摸点检测装置上接收触摸信息，并将它转换成为触点坐标，再送给 CPU 处理，它同时能接收 CPU 发来的命令并加以执行。

### 7.1.2　触摸屏的主要类型

按技术原理来区别触摸屏，可分为以下 5 个基本种类：

- 矢量压力传感式触摸屏；
- 电阻式触摸屏；
- 电容式触摸屏；

- 红外线式触摸屏；
- 表面声波式触摸屏。

其中矢量压力传感式触摸屏已退出历史舞台，这里不再进行介绍；红外线技术触摸屏价格低廉，但它的外框易碎，比较容易产生光干扰，曲面情况下失真；电容技术触摸屏设计构思合理，但其图像失真问题很难得到根本解决；电阻技术触摸屏的定位准确，但其价格颇高，且怕刮易损；表面声波触摸屏解决了以往触摸屏的各种缺陷，清晰且不容易被损坏，适用于各种场合，缺点是屏幕表面如果有水滴和尘土会使触摸屏变得迟钝，甚至不工作。根据触摸屏的工作原理和传输信息的介质，可以把触摸屏分为 4 种，它们分别为电阻式、电容感应式、红外线式以及表面声波式。每一类触摸屏都有它们各自的优缺点，了解哪种触摸屏适用于哪种场合，关键就在于要懂得每一类触摸屏技术的工作原理和特点。下面对后面 4 种类型的触摸屏进行简要介绍一下：

### 1．电阻触摸屏

电阻触摸屏的屏体部分是一块与显示器表面相匹配的多层复合薄膜，由一层有机玻璃作为基层，表面还涂有一层透明的导电层，上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层透明导电层，在两层导电层之间有许多细小（小于千分之一英寸）的透明隔离点把它们隔开绝缘，如图 7.1 所示。

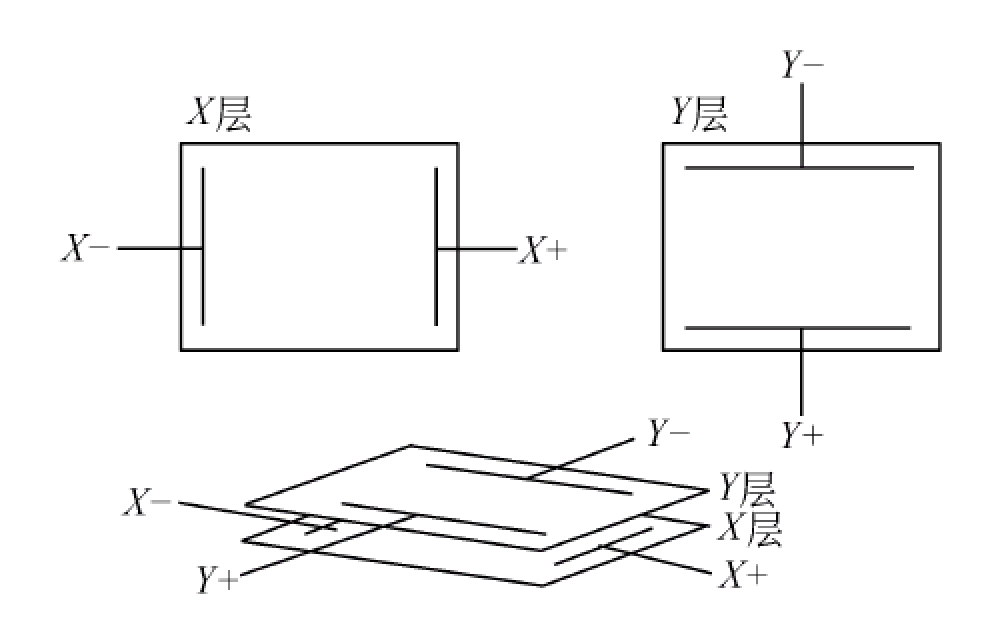



图 7.1　电阻触摸屏示意图

当手指点击屏幕时，平时相互绝缘的两层导电层就在触摸点位置产生了一个接触，因为其中一面导电层接通 Y 轴方向的 5V 均匀电压场，使得侦测层的电压由 0 变成了非零，这种接通状态被控制器侦测到后，进行 A/D 转换，并将得到的电压值与 5V 相比就可得到触摸点的 Y 轴坐标，同理得出 X 轴的坐标，这就是所有电阻技术触摸屏共同的最基本原理。电阻类触摸屏的关键是材料科技。电阻屏根据引出的线数多少，可分为四线、五线、六线等多线电阻触摸屏。电阻式触摸屏在强化玻璃表面分别涂上了两层 OTI 透明氧化金属导电层，在最外面的 OTI 涂层作为导电体，第二层 OTI 则经过精密的网络附上横竖两个方向的+5V 至 0V 的电压场，两层 OTI 之间用细小的透明隔离点隔开。当手指接触屏幕时，两层 OTI 导电层之间就会出现一个接触点，程序同时检测电压及电流，计算出触摸的位置，其反应速度为 10～20ms。

五线式电阻触摸屏的外层导电层使用的是延展性较好的镍金涂层材料，由于频繁触摸

外导电层，使用延展性好的镍金材料可以延长使用寿命，但是工艺成本较为高昂。镍金导电层虽然延展性较好，但是只能作为透明导体，不适合作为电阻触摸屏的工作面，因为它导电率高，而且金属做到厚度非常均匀不容易，不适合作电压分布层，只能作探层。

电阻式触摸屏工作在一种对外界完全隔离的环境，不怕灰尘及水汽，其可以用任何物体来触摸，可以用来写字画画，比较适合工业控制领域及办公室内有限人的使用。电阻式触摸屏共同的缺点是复合薄膜的外层采用塑胶材料，不清楚原理的人太用力或使用锐器触摸可能划伤整个触控屏而导致报废。不过在限度之内，划伤只是伤及外导电层，外导电层的划伤对于五线电阻触摸屏来说是没有关系的，而对四线电阻触摸屏来说是致命的。

### 2. 电容式触摸屏

电容式触摸屏的构造主要是在玻璃屏幕上镀一层透明的薄膜层，再在导体层外加上了一块保护玻璃，双玻璃设计能够很好地保护导体层及感应器。

电容式触摸屏在触摸屏的四边均镀上狭长的电极，在导电体内形成一个了低电压交流电场。用户接触屏幕时，由于人体电场，手指与导体层间会形成一个耦合电容，四个电极发出的电流会流向触点，而电流强弱跟手指到电极的距离成正比，位于触摸屏幕后的控制器便会计算电流的比例及强弱，准确算出触摸点的位置。电容触摸屏的双玻璃不仅能保护导体及感应器，更有效地防止了外在环境因素对触摸屏造成的影响，就算屏幕沾有污秽、尘埃或油渍，电容式触摸屏仍然可以准确地算出触摸位置。

电容式触摸屏在玻璃表面贴上一层透明的特殊金属导电物质。当手指接触在金属层上时，触点的电容就发生变化，使得与其相连的振荡器频率发生变化，通过测量频率变化可以确定触摸位置从而获得信息。由于电容随温度、湿度或接地情况的不同而变化，故其稳定性较差，经常会产生漂移现象。这种触摸屏适用于系统开发的调试阶段。

### 3. 红外线式触摸屏

这种触摸屏由装在触摸屏外框上的红外线发射与接收感测元件构成，在屏幕表面上形成红外线探测网，任何触摸物体都可以改变触点上的红外线而实现触摸操作。红外触摸屏不受电流、电压和静电干扰，适宜某些恶劣的环境条件。其主要优点是价格低廉、安装方便、不需要卡或其他任何控制器，可以用在各档次的计算机上。此外，由于没有电容充、放电过程，其响应速度比电容式快，但分辨率较低。

红外线触摸屏原理也比较简单，只是在显示屏上加上光点距架框，不需要在屏幕表面加上涂层或接驳控制器。光点距架框的四边排列了红外线发射管和接收管，在屏幕的表面形成了一个红外线网。用户以手指触摸屏幕某一点，就会挡住经过该位置的横竖两条红外线，计算机程序便可即时算出触摸点位置。因为红外触摸屏不受电流、电压和静电干扰，所以适宜某些恶劣的环境条件。其主要优点是价格低廉、安装方便、不需要卡或其他任何控制器，可以用在各档次的计算机上。但是，因为只是在普通屏幕增加了框架，所以在使用过程中框架四周的红外线发射管及接收管很容易损坏。

### 4. 表面声波触摸屏

表面声波是一种沿着介质表面传播的机械波。这种触摸屏由触摸屏、声波发生器、反

射器和声波接收器组成，其中，声波发生器能发送一种高频声波跨越屏幕的表面，当手指触到屏幕时，触点上的声波就被阻止，由此确定触点坐标的位置。表面声波触摸屏不受温度、湿度等环境因素影响，分辨率很高，有很好的防刮性，寿命长（5000 万次无故障）；透光率高（92%），能保持清晰透亮的图像质量；没有漂移，只需安装时一次校正；有第三轴（即压力轴）响应，最适合公共场所使用。表面声波触摸屏的触摸屏部分可以是一块平面、球面或是柱面的玻璃平板，安装在 CRT、LED、LCD 或是等离子显示器屏幕的前面。该玻璃平板只是一块纯粹的强化玻璃，区别于其有触摸屏技术是没有任何贴膜和覆盖层。玻璃屏的左上角和右下角各固定了竖直和水平方向的超声波发射换能器，右上角固定了两个相应的超声波接收换能器。玻璃屏的四个周边则刻有 45°角由疏到密间隔非常精密的反射条纹。

# 7.2　S3C2440 ADC 接口使用

写设备驱动程序一般都是和具体的处理器芯片和接口打交道，本章所使用的依然是 S3C2440 芯片。S3C2440 的触摸屏控制器是和 ADC（模数转换控制器）结合在一起的，因此本节将介绍 S3C2440 的 ADC 及其触摸屏接口。

## 7.2.1　S3C2440 触摸屏接口概述

S3C2440 具有 8 通道模拟输入的 10 位 CMOS 模数转换器（ADC），它将输入的模拟信号为 10 位的二数字码。在 2.5MHz 的 A/D 转换器时钟下，最大转化速率可以达到 500KSPS。A/D 器支持片上采样和保持功能，并支持掉电模式。

此外，S3C2440 的 AIN[7]和 AIN[5]用于连接触摸屏的模拟信号输入。触摸屏接口电路一般由触摸屏、4 个外部晶体管和 1 个外部电压源组成，如图 7.2 所示。

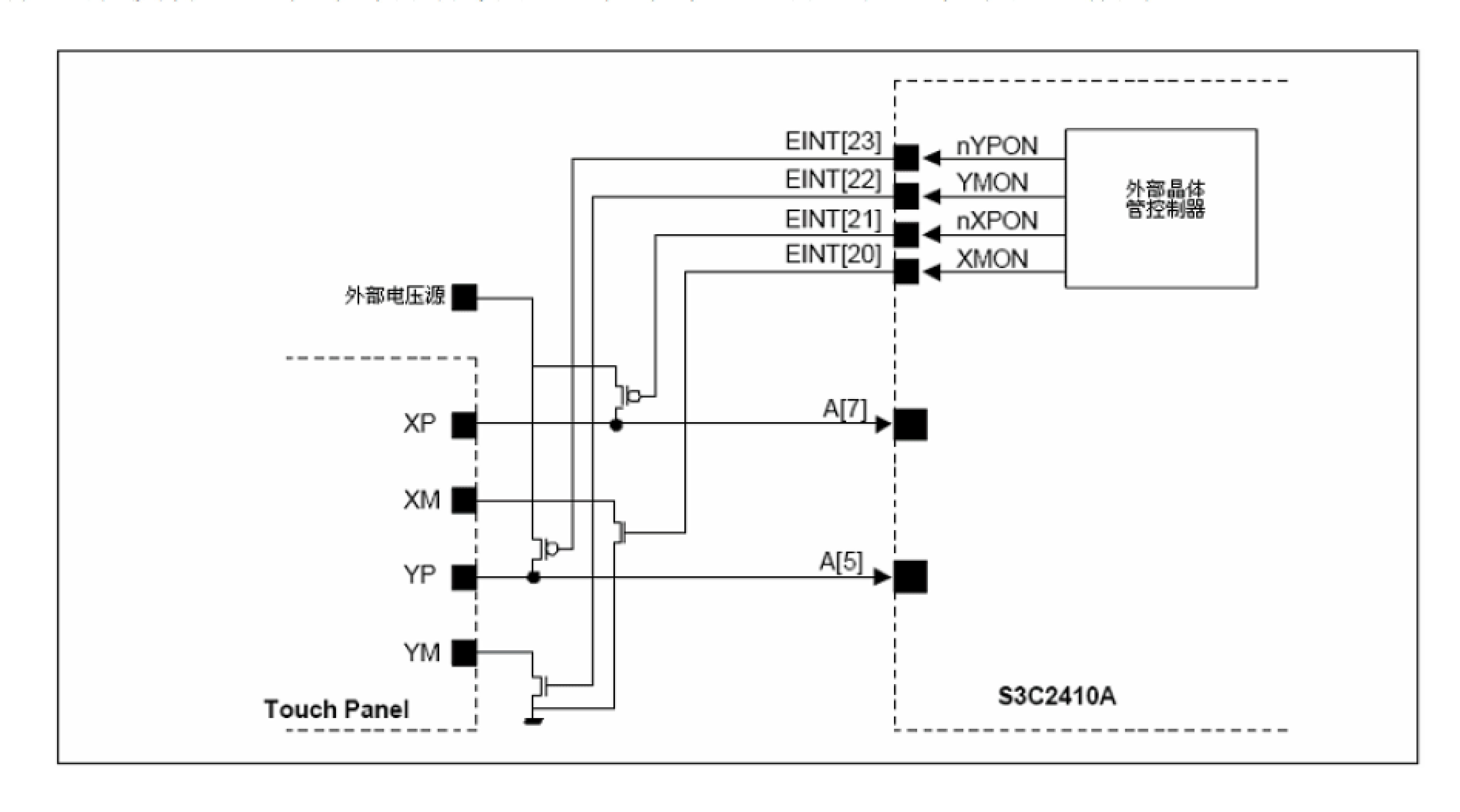



图 7.2　触摸屏接口电路示意图

触摸屏接口的控制和选择信号有 nYPON、YMON、nXPONT 和 XMON，它们连接切换 X 坐标和 Y 坐标转换的外部晶体管。模拟输入引脚（AIN[7]、AIN[5]）则连接到触摸屏引脚。

触摸屏控制接口包括一个外部晶体管控制逻辑和具有路数产生逻辑的 ADC 接口逻辑。

## 7.2.2　S3C2440 触摸屏接口操作

图 7.3 是 S3C2440 上的 A/D 转换器和触摸屏接口的功能框图。这个 A/D 转换器是一个循环类型的。上拉电阻接在 VDDA-ADC 和 AIN[7]之间。因此，触摸屏的 X+脚应该接到 S3C2440 的 AIN[7]，Y+脚则接到 S3C2440 的 AIN[5]。

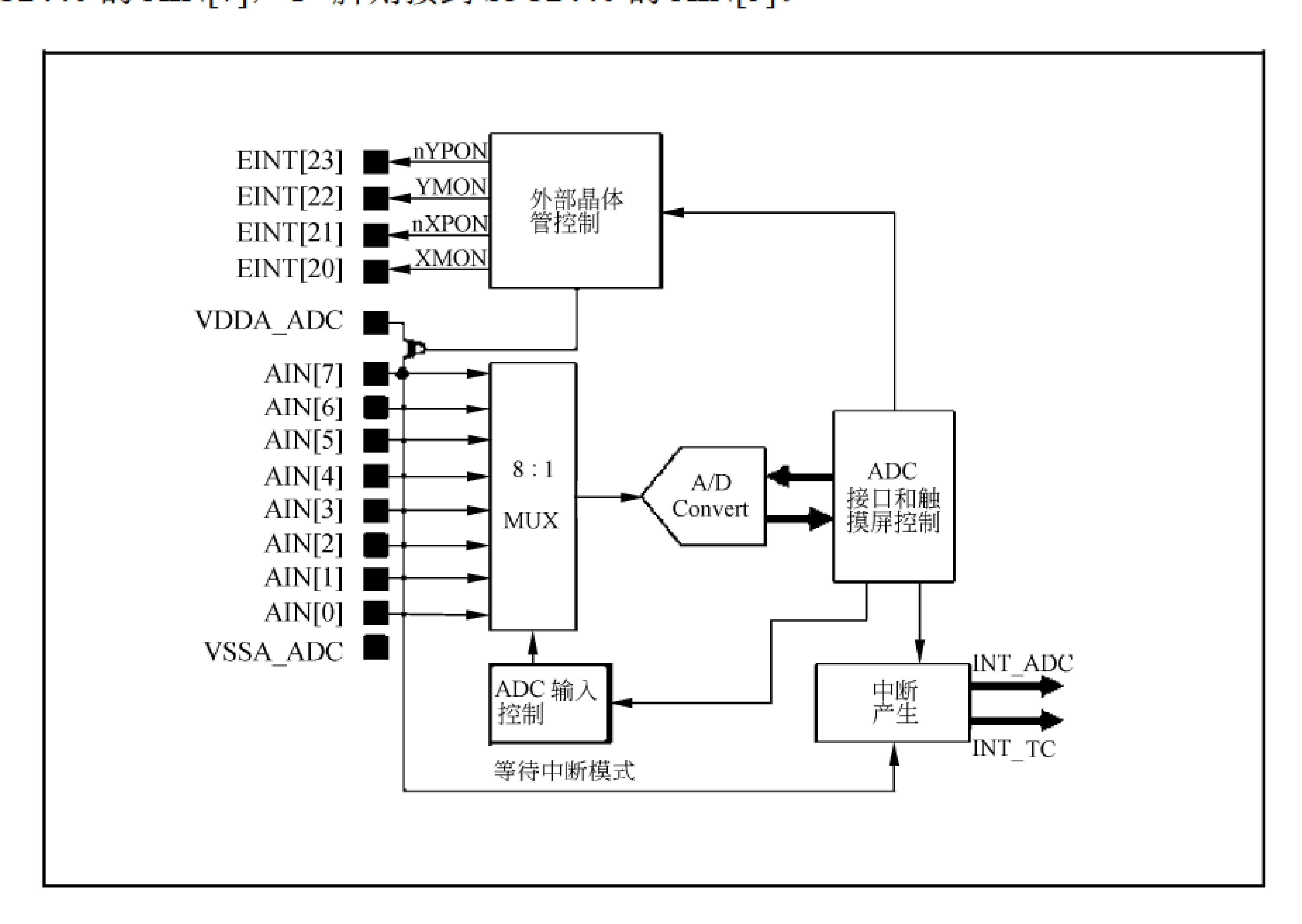



图 7.3　ADC 和触摸屏接口结构图

从图 7.3 可以知道，ADC 和触摸屏接口中只有一个 A/D 转换器，可以通过设置寄存器来选择对哪路模拟信号进行采样。图中有两个中断信号：INT_ADC 和 INT_TC，INT_ADC 表示 A/D 转换器已经转换完毕，而 INT_TC 则表示触摸屏被按下。

在使用触摸屏时，引脚 XP、XM、YP、YM 被用于和触摸屏直接相连，只剩下 AIN[3:0]共 4 个引脚用于一般的 ADC 输入；当不使用触摸屏时，XP、XM、YP 和 YM 这 4 个引脚也可以用于一般的 ADC 输入。

### 1．S3C2440触摸屏控制器工作模式

S3C2440 的触摸屏控制器是和 A/D 转换控制器结合在一起的，触笔的位置通过模拟信号传递给 A/D 转换器，A/D 转换完成后，把结果保存在相应的寄存器。根据转换方式的不同，触摸屏控制器有以下 4 种工作模式。

（1）等待中断模式

设置 ADCTSC 寄存器为 0XD3 即可令触摸屏控制器处于等待中断模式。这时，它在等待触摸屏被按下。当触摸屏被按下时，触摸屏控制器就发出 INT_TC 中断信号，这时触摸屏控制器要转入下面两种工作模式中的一种，来读取 x、y 方向的坐标。

- 当设置 ADCTSC 寄存器的位[8]为 0 时，表示等待 Pen Down 中断。
- 当设置 ADCTSC 寄存器的位[8]为 1 时，表示等待 Pen Up 中断。

等待中断模式下触摸屏引脚状况，如表 7.1 所示。

**表 7.1　等待中断模式下的触摸屏引脚状况**

| | XP | SM | YP | YM |
|---|---|---|---|---|
| 等待中断模式 | 上拉 | 高阻 | AIN[5] | 接地 |

（2）分离 x/y 轴坐标模式

设置 ADCTSC 寄存器为 0X69 进入 x 轴坐标转换模式，x 坐标值转换完毕后被写入 ADCDAT0，然后发出 INT_ADC 中断；同样地，设置 ADCDAT0 寄存器为 0X9A 进入 y 轴坐标转换模式，y 坐标值转换完毕后被写入 ADCDAT1，接着发出 INT_ADC 中断信号。分离 x/y 轴坐标模式下触摸屏引脚状况如表 7.2 所示。

**表 7.2　分离x/y轴坐标模式下的触摸屏引脚状况**

| | XP | SM | YP | YM |
|---|---|---|---|---|
| X | 接到外部电压 | 接地 | 接 AIN[5] | 高阻 |
| Y | 接 AIN[7] | 高阻 | 接外部电压 | 接地 |

（3）自动 x/y 轴坐标转换模式

设置 ADCTSC 寄存器为 0X0C，则进入自动 x/y 轴坐标转换模式，触摸屏控制器会自动转换触点的 x/y 坐标值然后分别保存在 ADCDAT0 和 ADCDAT1 寄存器中，之后发出 INT_ADC 中断信号。自动 x/y 轴坐标转换模式下的触摸屏引脚状况如表 7.3 所示。

**表 7.3　自动x/y轴坐标转换模式下的触摸屏引脚状况**

| | XP | SM | YP | YM |
|---|---|---|---|---|
| X | 接到外部电压 | 接地 | 接 AIN[5] | 高阻 |
| Y | 接 AIN[7] | 高阻 | 接外部电压 | 接地 |

（4）普通转换模式

这是一种普通的 A/D 转换，在不使用触摸屏时，触摸屏控制器处于这种工作模式。在这种模式下，可以通过设置 ADCCON 寄存器启动普通的 A/D 转换，转换结束后数据就被保存在 ADCDAT0 寄存器中。

### 2．S3C2440触摸屏接口专用寄存器

S3C2440 触摸屏接口涉及的专用寄存器比较少，主要有 ADCCON、ADCTSC、ADCDAT0 和 ADCDAT1。下面分别对它们进行介绍。

（1）ADCCON 控制寄存器。

它主要用来设置触摸屏的 A/D 转换方式，普通的 A/D 转换有 8 个输入通道，这 8 个通道是 AIN0～AIN7，其中 AIN5 和 AIN7 用于做为触摸屏的 x 和 y 方向的输入通道，具体描述如表 7.4 所示。

表 7.4　ADCCON控制寄存器描述

| ADCCON | 位 | 描　述 |
|---|---|---|
| DCFLG | [15] | A/D 转换结束标志<br>0：正在转换；1：转换结束 |
| PRSCEN | [14] | A/D 转换预分频使能<br>0：不使能；1：使能 |
| PRSCVL | [13:6] | A/D 转换预分频器数值<br>数据值范围：1～255<br>当预分频为 N 时，则除数实际上为（N+1）<br>注意：ADC 频率应该设置成小于 PLCK 的 5 倍<br>（例如：如果 PCLK=10MHz，ADC 频率小于 2 MHz） |
| SEL_MUX | [5:3] | 选择模拟输入通道<br>000：AIN0；001：AIN1<br>010：AIN2；011：AIN3<br>100：AIN4；101：AIN5<br>110：AIN6；111：AIN7 |
| STDBM | [2] | 选择静止模式<br>0：正常模式；1：静止模式 |
| READ_START | [1] | 通过读取来启动 A/D 转换<br>0：不启动；1：启动 |
| ENABLE_START | [0] | 通过设置该位来启动 A/D 操作。如果 READ_START 是使能的，这个值就是无效的<br>0：无操作；1：A/D 转换启动，启动后该位被清 0 |

（2）ADC 触摸屏控制寄存器 ADCTSC。

它主要用来选择触摸屏的工作模式。通过设置 ADCTSC 控制寄存器的值来设置触摸屏引脚状况从而可以设置触摸屏的工作模式，ADCTSC 控制寄存器的具体描述如表 7.5 所示。

表 7.5　ADCTSC控制寄存器描述

| ADCTSC | 位 | 描　述 |
|---|---|---|
| Reserved | [8] | 此位表示将检测哪类中断<br>0：按下；1：松开 |
| YM_SEN | [7] | 选择 YMON 的输出值<br>0：YMON 输出是 0（YM=高阻）<br>0：YMON 输出是 1（YM=GND） |
| YP_SEN | [6] | 选择 nYPON 的输出值<br>0：nYPON 输出是 0（YP=外部电压）<br>0：nYPON 输出是 1（YP 连接到 AIN[5]） |
| XM_SEN | [5] | 选择 XMON 的输出值<br>0：XMON 输出是 0（XM=高阻）<br>0：XMON 输出是 1（XM=GND） |
| SP_SEN | [4] | 选择 nXPON 的输出值<br>0：nXPON 输出是 0（XP=外部电压）<br>0：nXPON 输出是 1（XP 连接到 AIN[7]） |
| PULL_UP | [3] | 上拉切换使能<br>0：XP 上拉使能<br>1：XP 上拉禁止 |

续表

| ADCTSC | 位 | 描　　述 |
|---|---|---|
| AUTO_PST | [2] | 自动连续转换 X、Y 轴坐标<br>0：普通 ADC 转换<br>1：自动转换 |
| XY_PST | [1:0] | 手动测量 X、Y 轴坐标<br>00：无操作模式；01：对 X 轴坐标进行测量<br>10：对 Y 轴坐标测量；11：等待中断模式 |

（3）ADCDAT0 和 ADCDAT1 寄存器

这两个寄存器用来设置要转换的坐标和保存坐标的转换结果。X 轴的坐标转换结果会写到 ADCDAT0 寄存器的 XPDAT 中，等待转换完成后，触摸屏控制器会产生相应的中断；Y 轴的坐标转换结果会写到 ADCDAT1 寄存器的 YPDAT 中，等待转换完成后，触摸屏控制器会产生相应的中断，ADCDAT0 和 ADCDAT1 寄存器的具体描述如表 7.6 和表 7.7 所示。

**表 7.6　ADCDAT0 寄存器描述**

| ADCDAT0 | 位 | 描　　述 |
|---|---|---|
| UPDOWN | [15] | 等待中断模式下触笔的点击或抬起状态<br>0：触笔按下状态<br>1：触笔抬起状态 |
| AUTO_PST | [14] | 自动 X/Y 轴坐标转换模式<br>0：普通 ADC 转换<br>1：X/Y 轴坐标转换 |
| XY_PST | [13:12] | 手动 X/Y 坐标转换模式<br>00：无操作；01：X 轴坐标转换<br>10：Y 轴坐标转换；11：等待中断模式 |
| 保留 | [11:10] | 保留 |
| XPDATA | [9:0] | X 坐标的转换数据值 |

**表 7.7　ADCDAT1 寄存器描述**

| ADCDAT1 | 位 | 描　　述 |
|---|---|---|
| UPDOWN | [15] | 等待中断模式下触笔的点击或抬起状态<br>0：触笔按下状态<br>1：触笔抬起状态 |
| AUTO_PST | [14] | 自动 X/Y 轴坐标转换模式<br>0：普通 ADC 转换<br>1：X/Y 轴坐标转换 |
| XY_PST | [13:12] | 手动 X/Y 坐标转换模式<br>00：无操作；01：X 轴坐标转换<br>10：Y 轴坐标转换；11：等待中断模式 |
| 保留 | [11:10] | 保留 |
| YPDATA | [9:0] | Y 坐标的转换数据值 |

（4）ADC 起始延迟寄存器（ADCDLY）

ADCDLY 只有前 16 位有效，在正常转换模式，独立 X/Y 位置转换模式和自动 X/Y 位

置转换模式下，X/Y 位置转换延迟值；当在等待中断模式中有触笔按下时，这个寄存器在间歇的几毫秒时间内，为自动 X/Y 位置转换产生中断信号（INT_TC），好处在于在等待中断的时候还可以进行 AD 转换。

## 7.3　2.6 内核触摸屏驱动源码分析（s3c2410_ts.c 源码分析）

Linux 2.6.25 的内核源码中已经包含了触摸屏的相驱动了，驱动开发人员只要了解了内核的 LCD 驱动体系结构，然后参考内核中已有 LCD 驱动源码，再针对具体的触摸屏型号和硬件资源做相关修改就可以了。因此，本节开始将要分析 Linux 2.6.25 的触摸驱动源码，对应的源代码在 drivers/input/touchscreen/s3c2410_ts.c 中，这是一个对应于 S3C2410 片的触摸屏驱动代码，只要稍做修改就可以在 S3C2440 芯片上使用了。

文件 drivers/input/touchscreen/s3c2410_ts.c 是内核针对 S3C2410 芯片而设计的驱动程序，这个驱动程序也是一个平台驱动结构，通过向内核注册 device_driver 结构初始化触摸屏。device_driver 结构必须实现两个函数，分别是 probe 和 remove，在这里分别对应于 s3c2410ts_probe 和 s3c2410ts_remove。s3c2410ts_probe 是一个初始化函数，主要功能是完成资源的获取和对硬件初始化，以下分别介绍。

### 1. s3c2410ts_probe分析

s3c2410ts_probe 是一个探测函数，在这个函数中完成了硬件资源获取、GPIO 口的初始化、中断申请和注册驱动程序等操作。下面跟着程序流程一步步分析。

```
static int __init s3c2410ts_probe(struct device *dev)
{
/*结构 s3c2410_ts_mach_info 用于保存触摸屏的特定数据，里面存放的是触摸屏需要的一些设置参数，如分频比和延时等参数*/
    struct s3c2410_ts_mach_info *info;

    info = ( struct s3c2410_ts_mach_info *)dev->platform_data;
                                //从传进来的平台数据中获取硬件特定数据

    if (!info)
    {
        printk(KERN_ERR "Hm... too bad : no platform data for ts\n");
        return -EINVAL;             //出错返回
    }

#ifdef CONFIG_TOUCHSCREEN_S3C2410_DEBUG
    printk(DEBUG_LVL "Entering s3c2410ts_init\n"); //这是一些调试信息的打印
#endif

    adc_clock = clk_get(NULL, "adc");
        /*获取时钟，挂载 APB BUS 上的外围设备，需要时钟控制，ADC 就是这样的设备*/
    if (!adc_clock) {
        printk(KERN_ERR "failed to get adc clock source\n");
        return -ENOENT;
    }
    //clk_use(adc_clock);
    clk_enable(adc_clock);
```

```
#ifdef CONFIG_TOUCHSCREEN_S3C2410_DEBUG
    printk(DEBUG_LVL "got and enabled clock\n");
#endif

/*开始映射 I/O 内存，I/O 内存是不能直接进行访问的，必须对其进行映射，为 I/O 内存分配虚
拟地址，这些虚拟地址以__iomem 进行说明，但不能直接对其进行访问，需要使用专用的函数，如
iowrite320()*/
    base_addr=ioremap(S3C2410_PA_ADC,0x20);
    if (base_addr == NULL) {
        printk(KERN_ERR "Failed to remap register block\n");
        return -ENOMEM;
    }

    /* Configure GPIOs */
    s3c2410_ts_connect();               //设置触摸屏相关的 4 个 GPIO 为特殊功能

    if ((info->presc&0xff) > 0)
        writel(S3C2410_ADCCON_PRSCEN | S3C2410_ADCCON_PRSCVL(info->presc
        &0xFF),\
                base_addr+S3C2410_ADCCON); //使能预分频和设置分频系数
    else
        writel(0,base_addr+S3C2410_ADCCON);


    /* Initialise registers */
    if ((info->delay&0xffff) > 0)
        writel(info->delay & 0xffff,  base_addr+S3C2410_ADCDLY);
                    /*设置 ADC 延时，在等待中断模式下表示产生 INT_TC 的间隔时间*/

    writel(WAIT4INT(0), base_addr+S3C2410_ADCTSC);
                    //按照等待中断的模式设置 TSC
```

下面开始初始化 s3c2410ts 结构体，s3c2410ts 结构体定义如下：

```
struct s3c2410ts {
        struct input_dev dev;       //定义输入设备
        long xp;                    //X 轴坐标
        long yp;                    //Y 轴坐标
        int count;                  //统计采样次数
        int shift;                  //用于根据采样次数求平均值
        char phys[32];              //设备名称
};
    memset(&ts, 0, sizeof(struct s3c2410ts));
    init_input_dev(&ts.dev);
```

设置事件类型，下面几句都是设置事件类型中的代码，要理解这些代码，需先理解事件类型，常用的事件类型有 EV_KEY、EV_MOSSE，EV_ABS（用来接收像触摸屏这样的绝对坐标事件），而每种事件又会有不同类型的编码 code，比如 ABS_X，ABS_Y，这些编码又会有相应的值，关于内核的输入子系统在后面详细介绍。

```
    ts.dev.evbit[0]  =  ts.dev.evbit[0]  =  BIT(EV_SYN)  |  BIT(EV_KEY)  |
    BIT(EV_ABS);
    ts.dev.keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);
    input_set_abs_params(&ts.dev, ABS_X, 0, 0x3FF, 0, 0);
    input_set_abs_params(&ts.dev, ABS_Y, 0, 0x3E8, 0, 0);
    input_set_abs_params(&ts.dev, ABS_PRESSURE, 0, 1, 0, 0);
```

```
    sprintf(ts.phys, "ts0");                    //填写设备名称
/*以上是输入设备的名称和 ID，这些信息是输入设备的身份信息*/
    ts.dev.private = &ts;
    ts.dev.name = s3c2410ts_name;
    ts.dev.phys = ts.phys;
    ts.dev.id.bustype = BUS_RS232;
    ts.dev.id.vendor = 0xDEAD;
    ts.dev.id.product = 0xBEEF;
    ts.dev.id.version = S3C2410TSVERSION;

    ts.shift = info->oversampling_shift;    //设置采样次数
```

下面开始注册中断处理进程。stylus_action 和 stylus_updown 是两个中断处理函数，当笔尖触摸时，会进入到 stylus_updown。这里申请了触摸屏相关的两个中断，一个是 IRQ_TC 中断，查阅了数据手册后了解到，这个中断在笔按下时，由 XP 管脚产生表示中断的低电平信号，而笔抬起是没有中断信号产生的。另一个是 IRQ_ADC_DONE 中断，该中断是当芯片内部 A/D 转换结束后，通知中断控制器产生中断，此时就可以去读取转换得到的数据。

```
    //if (request_irq(IRQ_ADC, stylus_action, SA_SAMPLE_RANDOM | SA_SHIRQ,
    if (request_irq(IRQ_ADC, stylus_action, SA_SAMPLE_RANDOM,
        "s3c2410 action", &ts.dev)) {
        printk(KERN_ERR "s3c2410_ts.c: Could not allocate ts IRQ_ADC !\n");
        iounmap(base_addr);
        return -EIO;
    }   //ADC 转换中断，转换结束后触发
    if (request_irq(IRQ_TC, stylus_updown, SA_SAMPLE_RANDOM,
            "s3c2410_action", &ts.dev)) {
        printk(KERN_ERR "s3c2410_ts.c: Could not allocate ts IRQ_TC !\n");
        iounmap(base_addr);
        return -EIO;
    }   //TSC 中断，触笔动作触发

    printk(KERN_INFO "%s successfully loaded\n", s3c2410ts_name);

    /* All went ok, so register to the input system */
    input_register_device(&ts.dev);           //注册输入设备
    printk(KERN_INFO "%s input_register_device\n", s3c2410ts_name);
    return 0;
}
```

## 2．touch_timer_fire分析

touch_timer_fire()函数主要实现以下功能：

- stylus down 的时候，touch_timer_fire()函数在中断函数 stylus_updown 里被调用，此时缓存区没有数据，ts.count 的值为 0，所以只是简单地设置 A/D 转换的模式，然后开启 A/D 转换。
- 当 ADC 中断函数 stylus_action()把缓冲区填满时，作为中断后半段函数稍后被调用，此时 ts.count 等于 shift，算出其平均值后，交给事件处理层（Event Handler）处理，主要是填写缓冲然后唤醒等待输入数据的进程。
- stylus 抬起，等到缓冲区填满后（可能会包含一些无用的数据）被调用，这时判断出 stylus up，报告 stylus up 事件，重新等待 stylus down。

下面分析具体代码。

```
static void touch_timer_fire(unsigned long data)
{
    unsigned long data0;        //保存 X 轴的坐标
    unsigned long data1;        //保存 Y 轴的坐标
    int updown;                 //保存触笔按下或抬起，按下为 1

    data0 = readl(base_addr+S3C2410_ADCDAT0);      //读 X 轴的坐标
    data1 = readl(base_addr+S3C2410_ADCDAT1);      //读 Y 轴的坐标

    updown = (!(data0 & S3C2410_ADCDAT0_UPDOWN)) && (!(data1 & S3C2410_
    ADCDAT0_UPDOWN));                              //测试触笔是否按下

/*触笔按下后，开始对 X 和 Y 轴的坐标进行 A/D 转换，为了求得比较精确的坐标值，可以进行多
次转换后取平均值。这里的 ts.shift 就是转换（采样）的次数*/
    if (updown) {                                  //触笔按下
        if (ts.count != 0) {                       //多次采样已经完成
            long tmp;

            tmp = ts.xp;
            ts.xp = ts.yp;
            ts.yp = tmp;

            ts.xp >>= ts.shift;                    //求平均值
            ts.yp >>= ts.shift;

/*调试信息*/
#ifdef CONFIG_TOUCHSCREEN_S3C2410_DEBUG
        {
                struct timeval tv;
                do_gettimeofday(&tv);
                printk(DEBUG_LVL "T: %06d, X: %03ld, Y: %03ld\n", (int)tv.
                tv_usec, ts.xp, ts.yp);
        }
#endif

            /* 下面两句是报告 X、Y 的绝对坐标值 */
input_report_abs(&ts.dev, ABS_X, ts.xp);
            input_report_abs(&ts.dev, ABS_Y, ts.yp);

            /* 报告按键事件，键值为 1（代表触摸屏对应的按键被按下）*/
            input_report_key(&ts.dev, BTN_TOUCH, 1);

            /* 报告触摸屏的状态，1 表明触摸屏被按下 */
            input_report_abs(&ts.dev, ABS_PRESSURE, 1);

            /* 等待接收方收到数据后回复确认，用于同步 */
            input_sync(&ts.dev);
        }
/*如果触笔是刚刚按下的，那么 ts.count 的值为 0，此时要清空之前保存的数值。有一种情况，
当触笔在屏幕上拖动时，会不停地采样并报告坐标值*/
        ts.xp = 0;
        ts.yp = 0;
        ts.count = 0;

        writel(S3C2410_ADCTSC_PULL_UP_DISABLE | AUTOPST, base_addr+S3C
        2410_ADCTSC);  //设置自动转换
        writel(readl(base_addr+S3C2410_ADCCON) | S3C2410_ADCCON_ENABLE_
```

```
        START, base_addr+S3C2410_ADCCON);      //开始 A/D 转换
    } else {    //这种情况是触笔抬起的时候

        ts.count = 0;

         /* 报告按键事件，键值为 1（代表触摸屏对应的按键被释放）*/
        input_report_key(&ts.dev, BTN_TOUCH, 0);

        /* 报告触摸屏的状态，0 表明触摸屏没被按下 */
        input_report_abs(&ts.dev, ABS_PRESSURE, 0);
        input_sync(&ts.dev);

        /* 进入 s3c2410 触摸屏提供的等待中断模式，等待触笔按下 */
        writel(WAIT4INT(0), base_addr+S3C2410_ADCTSC);
    }
}
```

### 3．stylus_updown分析

当有触笔按下屏幕时，触摸屏会产生触摸屏中断，这时函数 stylus_updown()就会被调用，从而进入中断服务，这是在 s3c2410ts_probe()函数设置中断调用时就已设置好了的。stylus_updown 主要完成触笔动作的检查工作，具体分析如下：

```
static irqreturn_t stylus_updown(int irq, void *dev_id, struct pt_regs
*regs)
{
    unsigned long data0;          //用于保存 ADCDAT0 的值
    unsigned long data1;          //用于保存 ADCDAT1 的值
    int updown;                   //用于保存触笔动作

    data0 = readl(base_addr+S3C2410_ADCDAT0);  //读 ADCDAT0 的值
    data1 = readl(base_addr+S3C2410_ADCDAT1);  //读 ADCDAT1 的值

/*再次判断触笔是否真的按下，本来进入这个中断服务程序就说明触笔是已经按下了，这里有延时
去抖动的作用*/
    updown = (!(data0 & S3C2410_ADCDAT0_UPDOWN)) && (!(data1 & S3C2410_
    ADCDAT0_UPDOWN));

    /* TODO we should never get an interrupt with updown set while
     * the timer is running, but maybe we ought to verify that the
     * timer isn't running anyways. */

    if (updown)
        touch_timer_fire(0);
        /*判断出 stylus down，调用 touch_timer_fire 函数，从而进入中断的底半部*/

    return IRQ_HANDLED;
}
```

### 4．stylus_action分析

这是针对 A/D 转换完成的中断处理函数，当 A/D 转换完成时，判断采样是否完成，没完成则继续采样，用 mod_timer 定时器来设置多次采样的时间间隔，具体分析如下：

```
static irqreturn_t stylus_action(int irq, void *dev_id, struct pt_regs
*regs)
```

```
{
    unsigned long data0;                            //用于保存 X 坐标的值
    unsigned long data1;                            //用于保存 Y 坐标的值

    data0 = readl(base_addr+S3C2410_ADCDAT0);       //读 X 坐标的值
    data1 = readl(base_addr+S3C2410_ADCDAT1);       //读 Y 坐标的值

    ts.xp += data0 & S3C2410_ADCDAT0_XPDATA_MASK;  //X 坐标的值累加
    ts.yp += data1 & S3C2410_ADCDAT1_YPDATA_MASK;  //Y 坐标的值累加
    ts.count++;                                     //计数器递增

       if (ts.count < (1<<ts.shift)) {              //判断是否完成采样次数
        writel(S3C2410_ADCTSC_PULL_UP_DISABLE | AUTOPST, base_addr+S3C24
        10_ADCTSC);                                 //设置为自动模式
        writel(readl(base_addr+S3C2410_ADCCON) | S3C2410_ADCCON_ENABLE_
        START, base_addr+S3C2410_ADCCON);           //开始 A/D 转换
    } else {
        mod_timer(&touch_timer, jiffies+1);
                              //采样完成，调用 touch_timer_fire 报告坐标值
        writel(WAIT4INT(1), base_addr+S3C2410_ADCTSC);
    }

    return IRQ_HANDLED;
}
```

### 5．3c2410ts_remove分析

这是一个设备移除函数，当注销时该函数会被调用，它主要完成资源的释放，代码分析如下：

```
static int s3c2410ts_remove(struct device *dev)
{
    disable_irq(IRQ_ADC);                   //禁止 A/D 中断
    disable_irq(IRQ_TC);                    //禁止触摸屏中断
    free_irq(IRQ_TC,&ts.dev);               //释放触摸屏中断号
    free_irq(IRQ_ADC,&ts.dev);              //释放 A/D 中断号

    if (adc_clock) {
        clk_disable(adc_clock);             //关闭时钟
        //clk_unuse(adc_clock);
        clk_put(adc_clock);
        adc_clock = NULL;
    }

    input_unregister_device(&ts.dev);   //注销输入设备
    iounmap(base_addr);

    return 0;
}
```

## 7.4　Linux 内核输入子系统介绍

前面分析了 S3C2410 的触摸屏驱动，那么，这里的驱动是怎么和应用程序交互的呢？

这中间就用到 Linux 的输入子系统了，现在深入到下一层，对 Linux 的输入子系统进行分析。

## 7.4.1　Input 子系统概述

Linux 系统提供了 Input 子系统，输入子系统由输入子系统核心层（input core）、驱动层和事件处理层（event handler）3 部分组成。输入事件（如鼠标移动、键盘按键按下、joystick 的移动等）通过 driver ->inputcore -> eventhandler -> userspace 的顺序到达用户空间传给应用程序。按键、触摸屏、键盘、鼠标等输入都可以利用 Input 接口函数来实现设备驱动。

在 Linux 内核中，Input 设备用 input_dev 结构体描述，使用 Input 子系统实现输入设备驱动程序的时候，驱动的核心工作是向系统报告按键、触摸屏、键盘、鼠标等输入事件（event，通过 input_event 结构体描述），不需要再关心文件操作接口，因为 Input 子系统已经完成了文件操作接口。驱动报告的事件经过 InputCore 和 Eventhandler 最终到达用户空间。例如，触摸屏将检测到的所有按键都上报给了 Input 子系统。Input 子系统是所有 I/O 设备驱动的中间层，为上层提供了一个统一的接口界面。所以，在终端系统中，我们不需要去管有多少个键盘，多少个鼠标，它只要从 Input 子系统中去取对应的事件（按键、鼠标移位等）就可以了。

## 7.4.2　输入设备结构体

要了解输入设备子系统，就得先了解内核中输入设备的定义，这里先给出内核中 input_dev 的定义，然后再对其中重要的成员进行描述，内核中 input_dev 的定义如下：

```
struct input_dev {
    /* private: */
    void *private;  /* do not use */
    /* public: */

    const char *name;
    const char *phys;
    const char *uniq;
    struct input_id id;

    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];

    unsigned int keycodemax;
    unsigned int keycodesize;
    void *keycode;
    int (*setkeycode)(struct input_dev *dev, int scancode, int keycode);
    int (*getkeycode)(struct input_dev *dev, int scancode, int *keycode);
```

```
    struct ff_device *ff;

    unsigned int repeat_key;
    struct timer_list timer;

    int sync;

    int abs[ABS_MAX + 1];
    int rep[REP_MAX + 1];

    unsigned long key[BITS_TO_LONGS(KEY_CNT)];
    unsigned long led[BITS_TO_LONGS(LED_CNT)];
    unsigned long snd[BITS_TO_LONGS(SND_CNT)];
    unsigned long sw[BITS_TO_LONGS(SW_CNT)];

    int absmax[ABS_MAX + 1];
    int absmin[ABS_MAX + 1];
    int absfuzz[ABS_MAX + 1];
    int absflat[ABS_MAX + 1];

    int  (*open)(struct input_dev *dev);
    void (*close)(struct input_dev *dev);
    int  (*flush)(struct input_dev *dev, struct file *file);
    int (*event)(struct input_dev *dev, unsigned int type, unsigned int code,
    int value);

    struct input_handle *grab;

    spinlock_t event_lock;
    struct mutex mutex;

    unsigned int users;
    int going_away;

    struct device dev;

    struct list_head    h_list;
    struct list_head    node;
};
```

下面分析几个重要的字段，这是在驱动程序中经常用到的。

### 1．private字段

在 Input 结构中，这个字段可以被用来指向在输入设备驱动程序中的任何私有数据结构，例如在驱动处理多个设备时。在 open()和 close()函数中，需要此字段。

### 2．ID和name字段

在注册输入设备前，驱动程序应该设置 dev->name。它是一个字符串，例如 Generic button device，包含了一个设备的名字。

ID 字段包含了总线 ID（PCI、USB，...），供应商 ID 和设备的设备 ID。总线 ID 在 input.h 文件中定义。供应商和设备 ID 在 pci_ids.h、usb_ids.h 和相似的头文件中被定义。这些字段应该在注册输入设备前被驱动程序设置。

ID 和 name 字段可以通过 evdev 接口传递给用户空间使用。

### 3．keycode、keycodemax、keycodesize字段

这 3 个字段可以用于所有输入设备，被用来报告将产生的数据作为扫描码。如果不是所有的扫描码可以被自动识别所辨别，它们可能需要通过用户空间应用程序设置。这样 keycode 数组被用来映射扫描码到输入系统的键码。 keycodemax 包含了数组的大小。keycodesize 表示数组中数据的大小（单位为 bytes）。

### 4．evbit 、keybit、relbit、absbit字段

这几个字段是用于设置输入设备的事件类型，EV_KEY 是最简单的事件类型，用作按键的事件类型。这个事件通过下面函数报告给输入系统：

```
input_report_key(struct input_dev *dev, int code, int value)
```

可以通过文件 linux/input.h 了解所有允许的码值（从 0 到 KEY_MAX）。参数 value 为布尔值，也就是任何非零值意味着键被按下，0 值表示键被释放。只有在 value 不同于以前的 value 的，输入码产生事件。

除了 EV_KEY，还有两个基本的事件类型，即 EV_REL 和 EV_ABS。它们被用于表示设备提供的相对和绝对值，例如，一个相对值可以是鼠标在 X 轴上的移动距离。鼠标上报此值作为相对于此前最后位置的差值，因为鼠标没有采用任何绝对坐标系统。绝对事件用于摇杆和数字仪（joysticks and digitizers），此类设备在一个绝对坐标系统下工作。

让设备上报 EV_REL 和上报 EV_KEY 一样简单，只需设置对应位并调用下面函数，仅在为非零值产生事件报告。

```
input_report_rel(struct input_dev *dev, int code, int value)
```

然而，EV_ABS 需要一些特殊的处理。在调用 input_register_device()函数之前，必须为设备具有的每个绝对坐标填充 input_dev 数据结构中相关的字段。假如例子中的按键设备也有 ABS_X 轴：

```
button_dev.absmin[ABS_X] = 0;
button_dev.absmax[ABS_X] = 255;
button_dev.absfuzz[ABS_X] = 4;
button_dev.absflat[ABS_X] = 8;
```

这个设置可能对摇杆 X 轴是适合的，最小值为 0，最大值为 255，数据误差是±4，中心平滑位置为 8。

如果不需要 absfuzz 和 absflat，可以设置它们的值为 0。这样就表示数据是精确的，总是返回到中心位置。

除了介绍的事件类型，其他的事件类型还包括：

```
EV_LED: used for the keyboard LEDs.
EV_SND: used for keyboard beeps.
```

这两个非常类似于 EV_KEY 事件，但是报告方向是相反的，也就是说从系统到输入设备驱动。如果你的输入设备驱动能处理这些事件，则驱动程序必须在 evbit 字段设置相对应的位和设置回调函数：

```
button_dev.event = button_event;
int button_event(struct input_dev *dev, unsigned int type, unsigned int code,
int value);
{
    if (type == EV_SND && code == SND_BELL) {
        outb(value, BUTTON_BELL);
        return 0;
    }
    return -1;
}
```

这个回调例程可以在中断上下文或者底半部 BH 中调用，因此不能休眠，也不能花费太长时间去完成。

## 7.4.3　输入链路的创建过程

输入链路的创建过程主要包括硬件设备注册和 input handler 两部分。下面依次讲解。

### 1. 硬件设备的注册

驱动程序负责和底层的硬件设备打交道，将底层硬件对用户输入的响应转换为标准的输入事件以后再向上发送给 input core。驱动程序通过调用 input_register_device 函数和 input_unregister_device 函数来向输入子系统中注册和注销输入设备。

这两个函数调用的参数是一个 input_dev 结构，这个结构在 driver/input/input.h 中定义。驱动程序在调用 input_register_device 之前需要填充该结构中的部分字段。如 s3c2410ts_probe 函数中的如下代码：

```
init_input_dev(&ts.dev);
ts.dev.evbit[0] = ts.dev.evbit[0] = BIT(EV_SYN) | BIT(EV_KEY) | BIT(EV_ABS);
ts.dev.keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);
input_set_abs_params(&ts.dev, ABS_X, 0, 0x3FF, 0, 0);
input_set_abs_params(&ts.dev, ABS_Y, 0, 0x3E8, 0, 0);
input_set_abs_params(&ts.dev, ABS_PRESSURE, 0, 1, 0, 0);
sprintf(ts.phys, "ts0");   //填写设备名称
/*以上是输入设备的名称和 ID，这些信息是输入设备的身份信息了*/
ts.dev.private = &ts;
ts.dev.name = s3c2410ts_name;
ts.dev.phys = ts.phys;
ts.dev.id.bustype = BUS_RS232;
ts.dev.id.vendor = 0xDEAD;
ts.dev.id.product = 0xBEEF;
ts.dev.id.version = S3C2410TSVERSION;
```

### 2. 注册input handler

驱动程序只是把输入设备注册到输入子系统中，在驱动层的代码中本身并不创建设备结点。应用程序用来与设备打交道的设备结点的创建由 event handler 层调用 input core 中的函数来实现。在创建具体的设备结点之前，event handler 层需要先注册一类设备的输入事件处理函数及相关接口，以 endev handler 为例，代码在 evdev.c 文件中可以找到：

```
static struct input_handler evdev_handler = {
    .event      = evdev_event,
```

```
    .connect     = evdev_connect,
    .disconnect = evdev_disconnect,
    .fops        = &evdev_fops,
    .minor       = EVDEV_MINOR_BASE,
    .name        = "evdev",
    .id_table    = evdev_ids,
};

static int __init evdev_init(void)
{
    return input_register_handler(&evdev_handler);
}
```

在 evdev_init 中调用 input.c 中定义的 input_register_handler 来注册一个事件类型的 handler。这里的 handler 不是具体用户可以操作设备，而是事件类设备统一的处理函数接口，如我们所说的触摸屏就是这种类型的设备。

总而言之，整个流程是硬件驱动向 input 子系统注册一个硬件设备后，在 input_register_device 中调用已经注册的所有类型的 input handler 的 connect()函数，每一个具体的 connect()函数会根据注册设备所支持的事件类型判断是否与自己相关，如果相关就调用 input_register_minor()创建一个具体的设备结点。

```
void input_register_device(struct input_dev *dev)
{
...
while (handler) {
 if ((handle = handler->connect(handler, dev)))
  input_link_handle(handle);
 handler = handler->next;
}
}
```

此外，如果已经注册了一些硬件设备，此后再注册一类新的 input handler，则同样会对所有已注册的 device 调用新的 input handler 的 connect()函数以确定是否需要创建新的设备结点。

```
void input_register_handler(struct input_handler *handler)
{
...
while (dev) {
 if ((handle = handler->connect(handler, dev)))
  input_link_handle(handle);
 dev = dev->next;
}
}
```

从上面的分析中可以看到一类 input handler 可以和多个硬件设备相关联，创建多个设备结点。而一个设备也可能与多个 input handler 相关联，创建多个设备结点。

### 7.4.4　使用 Input 子系统

在内核自带的文档 Documentation/input/input-programming.txt 中。有一个使用了 Input 子系统的例子，并附带相应的说明。下面以这个为例来分析如何使用 Input 子系统。

```
#include <linux/input.h>
```

```
#include <linux/module.h>
#include <linux/init.h>

#include <asm/irq.h>
#include <asm/io.h>

static void button_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
        input_report_key(&button_dev, BTN_1, inb(BUTTON_PORT) & 1);
                                        //报告按键事件
        input_sync(&button_dev);        //等待接收方收到数据后回复确认，用于同步
}

static int __init button_init(void)
{
        if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
                printk(KERN_ERR "button.c: Can't allocate irq %d\n",button_
                irq);
                return -EBUSY;
        }                                                   //注册中断处理函数

        button_dev.evbit[0] = BIT(EV_KEY);                  //设置输入设备是按键
        button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0);        //有一个按键

        input_register_device(&button_dev);                 //注册输入设备
}

static void __exit button_exit(void)
{
        input_unregister_device(&button_dev);               //注销输入设备
        free_irq(BUTTON_IRQ, button_interrupt);             //释放中断
}

module_init(button_init);
module_exit(button_exit);
```

这个示例 module 代码相对比较简单，在初始化函数里注册了一个中断处理例程。然后注册了一个 input device.在中断处理程序里，将接收到的按键上报到 Input 子系统。

首先，程序必须包含头文件<linux/input.h>，这个文件包含了输入子系统的接口，提供了输入设备所需要的所有定义。

在模块加载或内核启动时调用的_init()函数中，它申请了所需的资源（它也应该检查设备的存在）。

然后，它设置了输入位域。这是设备驱动告诉输入系统其他部分它是什么的方法，也就是说哪些事件可以被输入设备产生和接收。例子中设备仅仅产生 EV_KEY 类型事件，对应的事件码是 BTN_0。这样，可以仅仅设置这两位。也可以通过下面语句完成此功能，当需要设置多位时，例子中使用的方式更简洁。

```
set_bit(EV_KEY, button_dev.evbit);
set_bit(BTN_0, button_dev.keybit);
```

在设置完输入位域后，例子驱动通过语句 input_register_device(&button_dev);注册输入设备数据结构。它增加 button_dev 数据结构到输入驱动链表中，并调用设备处理模块_connect()函数去通知它们一个新的输入设备已经被发现。因为_connect()函数可能会调用

可以休眠的 kmalloc(, GFP_KERNEL)函数，所以 input_register_device()函数不能在中断上下文或者拥有自旋锁时调用。

在使用中，驱动中被调用的函数仅是 button_interrupt()。一旦从按键产生中断，此函数检查按键的状态，通过 input_report_key()函数调用向输入系统报告。在此，不需要检查中断函数是否通知了两个相同的事件值（例如，按下，按下），因为 input_report_*函数本身完成了此项功能。

通过 input_sync()函数调用通知事件的接收者，已经发送了完整的报告。在一个按键情况下，这个函数看起来并不重要。但在鼠标移动下，此函数就非常重要了。因为你不希望 X 值和 Y 值被分开解释，否则，它们将导致不同的鼠标移动事件。

### 7.4.5　编写输入设备驱动需要完成的工作

从上面这个例子可以看出，通过 Input 子系统，具体的输入设备驱动只需要完成如下工作：

#### 1．在模块加载函数中告知Input子系统它可以报告的事件

设备驱动通过 set_bit()告诉 Input 子系统它支持哪些事件，如下所示。

```
set_bit(EV_KEY, button_dev.evbit);
set_bit(BTN_0, button_dev.keybit);
```

这两个函数分别用来设置设备所产生的事件及上报的按键值。Struct iput_dev 中有两个成员，一个是 evbit、一个是 keybit，分别用于表示设备所支持的动作和按键类型。也可以像下面一样直接赋值：

```
button_dev.evbit[0] = BIT(EV_KEY);              //设置输入设备是按键
button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0);    //有一个按键
```

#### 2．在模块加载函数中注册输入设备

设备驱动可以通过 input_register_device()注册一个输入设备，函数原型如下：

```
input_register_device(&button_dev);
```

这个函数用来向内核注册一个输入设备，它的参数类型是一个输入设备结构体，驱动程序在初始化输入设备结构体后调用该函数进行注册。

#### 3．在键被按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时，通过input_report_xxx()函数报告发生的事件及对应的键值/坐标等状态

主要的事件类型包括 EV_KEY（按键事件）、EV_REL（相对值，如光标移动，报告的是相对最后一次位置的偏移）和 EV_ABS（绝对值，如触摸屏、操纵杆，它们工作在绝对坐标系统）。

用于报告 EV_KEY、EV_REL 和 EV_ABS 事件的函数分别为：

```
void input_report_key(struct input_dev *dev, unsigned int code, int value);
void input_report_rel(struct input_dev *dev, unsigned int code, int value);
```

```
void input_report_abs(struct input_dev *dev, unsigned int code, int value);
input_sync()
```

其中 input_sync 用来告诉上层，本次的事件已经完成了，用作同步。例如，在触摸屏设备驱动中，一次坐标及按下状态的报告过程如下：

```
input_report_abs(input_dev, ABS_X, x);              //X 坐标
input_report_abs(input_dev, ABS_Y, y);              //Y 坐标
input_report_abs(input_dev, ABS_PRESSURE, pres);    //压力
input_sync(input_dev);                              //同步
```

#### 4. 在模块卸载函数中注销输入设备

注销输入设备的函数如下：

```
void input_unregister_device(struct input_dev *dev);
```

是前一节内核源码的 3c2410ts_remove 函数中用下面函数注销触摸屏输入设备。

```
input_unregister_device(&ts.dev);                   //注销输入设备
```

## 7.5　触摸屏驱动移植和内核编译

在 Linux 2.6.25.8 中没有针对 S3C2440 芯片的触摸屏驱动程序，所以要在 S3C2440 芯片使用触摸屏必须自己编写触摸屏驱动程序，然而不可能从头编写，那样对项目开发进度来说是缓慢的。Linux 的好处是开源，所以可以参考 Linux 已经有的相关驱动程序做简单修改。

Linux 2.6.25.8 已经包含了针对 S3C2410 芯片的触摸屏驱动程序，前面也已经对该驱动源码进行了分析，这样，对于编写针对 S3C2440 芯片的触摸屏驱动程序就容易多了，简单的触摸屏驱动移植过程如下。

### 7.5.1　修改初始化源码

本节主要是几个关键文件，这些文件完成一些硬件初始化工作。读者看这里的时候可以对照内核原代码来看。

#### 1. 修改arch/arm/mach-s3c2440/mach-smdk2440.c

这个文件夹的上部分写的都是各个硬件设备的初始化数据。比如串口的初始化数据定义如下：

```
static struct s3c2410_uartcfg smdk2440_uartcfgs[] __initdata = {
    [0] = {
        .hwport      = 0,
        .flags       = 0,
        .ucon        = 0x3c5,
        .ulcon       = 0x03,
        .ufcon       = 0x51,
```

```
    },
    [1] = {
        .hwport      = 1,
        .flags       = 0,
        .ucon        = 0x3c5,
        .ulcon       = 0x03,
        .ufcon       = 0x51,
    },
    /* IR port */
    [2] = {
        .hwport      = 2,
        .flags       = 0,
        .ucon        = 0x3c5,
        .ulcon       = 0x43,
        .ufcon       = 0x51,
    }
};
```

该文件的上半部都是此类信息，这类信息多数是在该文件的下半部分会用得到的，对于我们所使用的 Linux 2.6.25.8 内核来说，已经有了触摸屏初始化数定义了，但如果是其他没有触摸屏初始化数定义的内核版本，则必须在这里加入如下代码：

```
static struct s3c2410_ts_mach_info s3c2440_ts_info = {
    .delay = 10000,                //延迟
    .presc = 49,                   //分频
    .oversampling_shift = 2,       //采样次数
};
```

在下面数组中加入关于触摸屏的平台数据，这里我们加入上面的 s3c_device_ts：

```
static struct platform_device *smdk2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,
    &s3c_device_ts, //这个在 devs.c 中定义，看下文
};
```

在下面代码段中加入黑体部分，用于初始化设备：

```
static void __init smdk2440_machine_init(void)
{
    s3c24xx_fb_set_platdata(&smdk2440_fb_info);
    s3c_device_ts.dev.platform_data = &s3c2440_ts_info;
    platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices));
    smdk_machine_init();
}
```

这样，arch/arm/mach-s3c2440/mach-smdk2440.c 文件部门的代码就算修改完成了。

### 2．修改arch/arm/plat-s3c24xx/devs.c

这个文件里面全是设备信息。这里常出现的就是 resource 类型的数据结构。代表各类资源，在这里加入下面代码，注意，要在文件尾，至少要在 s3c_adc_resource 定义之后，因为我们要用到它。

```
/*touch screen*/
struct platform device s3c device ts = {
    .name = "s3c2410-ts",
    .id = -1,
    .num resources = ARRAY SIZE(s3c adc resource),
.resource = s3c adc resource,
};
EXPORT_SYMBOL(s3c_device_ts)
```

对于上面的 EXPORT_SYMBOL 解释如下，当想用到 devs.c 中的数据结构时，应该怎么办？#include <devs.c>？没见过有人这样写。实际上真要在其他文件里用到 devs.c 中的东西，比如，我们想用到 s3c_device_ts，就在 devs.c 中写 EXPORT_SYMBOL(s3c_device_ts)；正如上面的代码一样，然后，在 devs.h 中写这么一句

```
extern struct platform_device s3c_device_ts
```

这样，在包含了头文件 devs.h 的 c 文件里，就可以使用这个通过 EXPORT_SYMBOL 导出来的变量了，正如我们前面 arch/arm/mach-s3c2440/mach-smdk2440.c 中的代码一样；如果 devs.c 中的东西是一个函数的话，我们还要通过“#include <devs.h>”的方式，只需要在 devs.h 中加入函数的原型即可，也不用 EXPORT_SYMBOL 这样的宏定义。

### 3．添加头文件

如果使用的内核版本里面没有 reg-adc.h，则需要从其他版本复制 reg-adc.h，文件位置在 include/asm-arm/arch-s3c2410/regs-adc.h，Linux 2.6.25.8 版本是有这个文件的，只要在该文件内添加如下内容，这些内容是用来设置触摸屏的工作模式的。

```
#define S3C2410 ADCTSC XY PST N     (0x0<<0)     //无操作模式
#define S3C2410_ADCTSC_XY_PST_X     (0x1<<0)     //对 X 坐标进行转换
#define S3C2410_ADCTSC_XY_PST_Y     (0x2<<0)         //对 Y 坐标进行转换
#define S3C2410_ADCTSC_XY_PST_W     (0x3<<0)         //等待中断模式
```

编写 s3c2440_ts.h 文件复制到 include/asm/arch-s3c2410/目录下，这个文件用于 S3C2440 触摸屏的平台数据结构，其内容如下所示。

```
#ifndef __ASM_ARM_S3C2440_TS_H
#define __ASM_ARM_S3C2440_TS_H
struct s3c2440 ts mach info {
      int           delay;                   //延迟
      int           presc;                   //预分频
      int           oversampling_shift;//采样次数
};
void __init set_s3c2440ts_info(struct s3c2440_ts_mach_info *hard_s3c2440-
ts_info);/*用于设置私有数据的函数*/
#endif /* __ASM_ARM_S3C2440_TS_H */
```

## 7.5.2　修改硬件驱动源码 s3c2440_ts.c

将下载的 S3C2410 的驱动程序代码复制到 drivers/input/touchscreen/ 下改名为 s3c2440_ts.c，因为 S3C2410 和 S3C2440 的触摸屏接口基本相同，所以不用对代码做多大修改，只要改变一下名字即可。

### 1．修改触摸屏的私有硬件结构体

这个结构体定义了 S3C2440 触摸屏的私有数据结构，包括坐标值和腰板计数器等信息，修改后的内容如下：

```
static char *s3c2440ts_name = "s3c2440 TouchScreen";
struct s3c2440ts {
 struct input_dev *dev;      //输入设备指针
 long xp;                    //保存 X 坐标
 long yp;                    //保存 Y 坐标
 int count;                  //采样记数器
 int shift;                  //要采样的次数
 char phys[32];
};
static struct s3c2440ts ts;//定义 S3C2440 触摸屏
```

### 2．初始化触摸屏并注册该触摸屏输入设备

注册一个输入设备之前要先对这个输入设备的结构体进行初始化，触摸屏是一种事件输入设备，这里把它同时初始化为一个同步事件、按键事件、绝对坐标事件，具体代码如下：

```
memset(&ts, 0, sizeof(struct s3c2440ts));
 ts.dev = input_allocate_device();          //分配一个输入设备到触摸屏结构中
...
 ts.dev->evbit[0]  =  ts.dev->evbit[0]  =  BIT(EV_SYN)  |  BIT(EV_KEY)  |
BIT(EV_ABS);/*同步事件、按键事件、绝对坐标事件*/
 ts.dev->keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);     //一个按键
 input_set_abs_params(ts.dev, ABS_X, 0, 0x3FF, 0, 0);  //绝对坐标的 X 方向
 input_set_abs_params(ts.dev, ABS_Y, 0, 0x3FF, 0, 0);  //绝对坐标的 Y 方向
 input_set_abs_params(ts.dev, ABS_PRESSURE, 0, 1, 0, 0);
                                            //按下还是抬起
 sprintf(ts.phys, "ts0");                   // dev 文件夹中该触摸屏的名字
 ts.dev->private = &ts;
 ts.dev->name = s3c2440ts_name;             //输入系统的名字
 ts.dev->phys = ts.phys;
 ts.dev->id.bustype = BUS_RS232;
 ts.dev->id.vendor = 0xDEAD;                //生产商代号
 ts.dev->id.product = 0xBEEF;
 ts.dev->id.version = S3C2440TSVERSION;
 ts.shift = info->oversampling_shift;
...
 printk(KERN_INFO "%s successfully loaded\n", s3c2440ts_name);
 /* All went ok, so register to the input system */
 input_register_device(ts.dev);             //注册 S3C2440ts 输入设备
```

### 3．注册S3C2440ts触摸屏驱动

下面是注册 S3C2440ts 触摸屏驱动的代码，系统初始化硬件设备时会调用 s3c2440ts_init()函数向系统注册这个驱动。

```
static struct device_driver s3c2440ts_driver = {
      .name          = "s3c2440-ts",
```

```
      .bus          = &platform_bus_type,
      .probe        = s3c2440ts_probe,
      .remove       = s3c2440ts_remove,
};

int __init s3c2440ts_init(void)
{
 return driver_register(&s3c2440ts_driver);
}
void __exit s3c2440ts_exit(void)
{
 driver_unregister(&s3c2440ts_driver);
}
```

## 7.5.3　修改 Kconfig 和 Makefile

要使驱动程序可以在内核的配置界面中显示并进行配置和编译，需要修改 drivers/input/touchscreen/Kconfig 和 drivers/input/touchscreen/Makefile。

### 1．修改drivers/input/touchscreen/Kconfig

在 drivers/input/touchscreen/Kconfig 文件中添加如下内容，这样在 make menuconfig 时就会在配置菜单中显示出 Samsung S3C2440 touchscreen input driver 这一项，选中它就可以编译驱动源码了。

```
config TOUCHSCREEN_S3C2440
 tristate "Samsung S3C2440 touchscreen input driver"
 depends on ARCH_S3C2440 && INPUT && INPUT_TOUCHSCREEN
 select SERIO
 help
   Say Y here if you have the s3c2440 touchscreen.
   If unsure, say N.
   To compile this driver as a module, choose M here: the
   module will be called s3c2440_ts.
config TOUCHSCREEN_S3C2440_DEBUG
 boolean "Samsung S3C2440 touchscreen debug messages"
 depends on TOUCHSCREEN_S3C2440
 help
   Select this if you want debug messages
```

### 2．修改drivers/input/touchscreen/Makefile

在 drivers/input/touchscreen/Makefile 中加入要编译进内核的驱动源码，这样就可以把上面的触摸屏驱动编译进内核中了，修改后结果如下，黑体为加入的内容。

```
#
# Makefile for the touchscreen drivers.
#

# Each configuration option enables a list of files.

obj-$(CONFIG_TOUCHSCREEN_ADS7846)       += ads7846.o
obj-$(CONFIG_TOUCHSCREEN_BITSY)         += h3600_ts_input.o
obj-$(CONFIG_TOUCHSCREEN_CORGI)         += corgi_ts.o
obj-$(CONFIG_TOUCHSCREEN_GUNZE)         += gunze.o
```

```
obj-$(CONFIG_TOUCHSCREEN_ELO)            += elo.o
obj-$(CONFIG_TOUCHSCREEN_FUJITSU)        += fujitsu_ts.o
obj-$(CONFIG_TOUCHSCREEN_MTOUCH)         += mtouch.o
obj-$(CONFIG_TOUCHSCREEN_MK712)          += mk712.o
obj-$(CONFIG_TOUCHSCREEN_HP600)          += hp680_ts_input.o
obj-$(CONFIG_TOUCHSCREEN_HP7XX)          += jornada720_ts.o
obj-$(CONFIG_TOUCHSCREEN_USB_COMPOSITE)     += usbtouchscreen.o
obj-$(CONFIG_TOUCHSCREEN_PENMOUNT)       += penmount.o
obj-$(CONFIG_TOUCHSCREEN_TOUCHRIGHT)     += touchright.o
obj-$(CONFIG_TOUCHSCREEN_TOUCHWIN)       += touchwin.o
obj-$(CONFIG_TOUCHSCREEN_UCB1400)        += ucb1400_ts.o
obj-$(CONFIG_S3C2410_TOUCHSCREEN)        += s3c2440_ts.o
```

## 7.5.4　配置编译内核

接下来就可以进入内核配置界面把驱动编译入内核中了。

### 1．把触摸屏驱动编译进内核

因为本章中在写触摸屏驱动程序时把触摸屏当成一种输入设备来设计的，所以在这里要进入 Device Drivers->Input device support 选项中，然后选上 Samsung S3C2440 touchscreen input driver 和 Samsung S3C2440 touchscreen debug messages 这两个选项，选前一个选项的目的是为了把触摸屏驱动源码选择编译进内核中，选后一个选项的目的是为了在标准输出上输出驱动代码中的调试信息，选择的结果如图 7.4 所示。

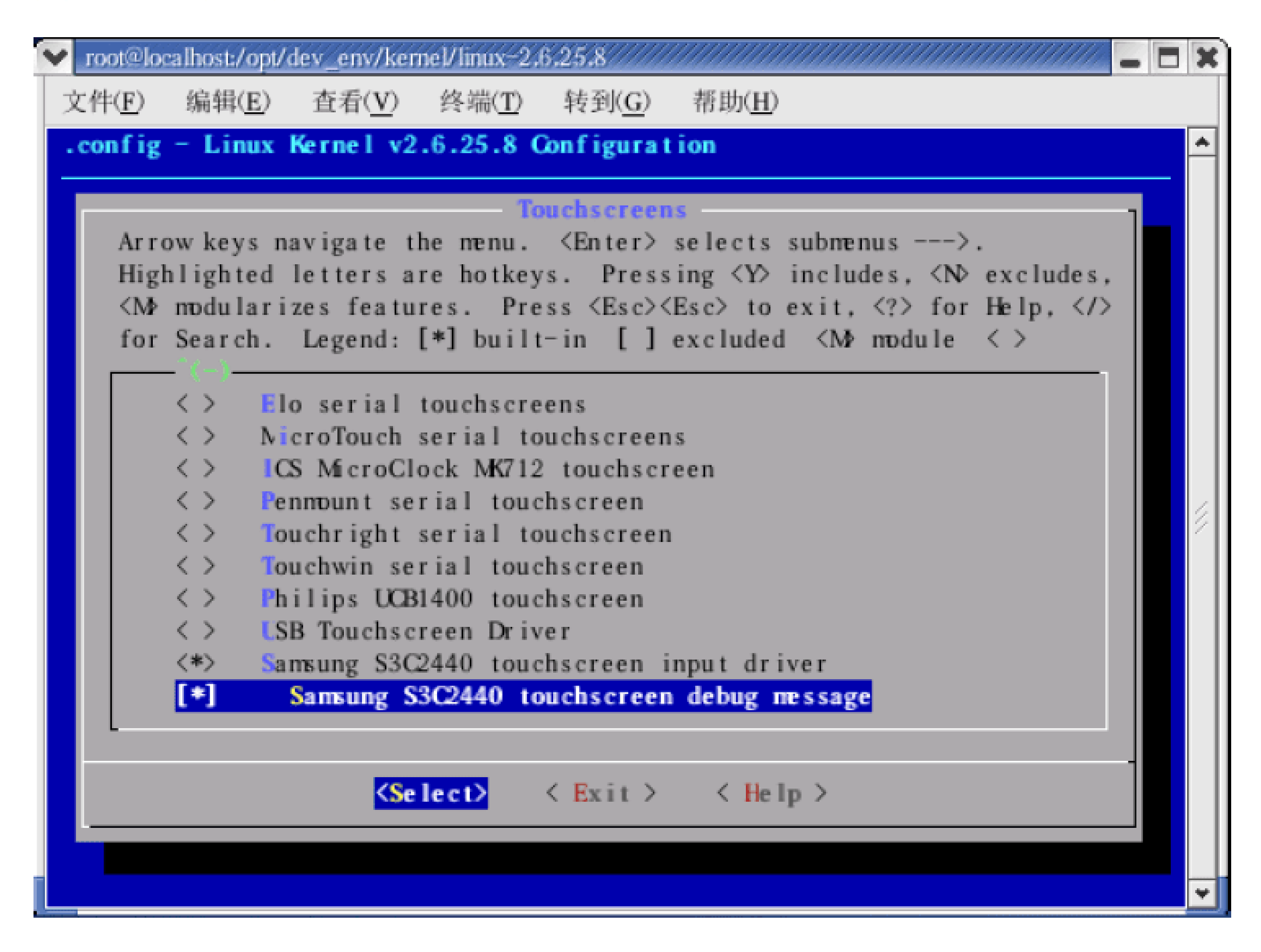



图 7.4　把触摸屏驱动编译进内核

### 2．选择Event debugging进行内核调试

选择 Event debugging 进行内核调试，当调试没错误后可以取消该选项，如图 7.5 所示。

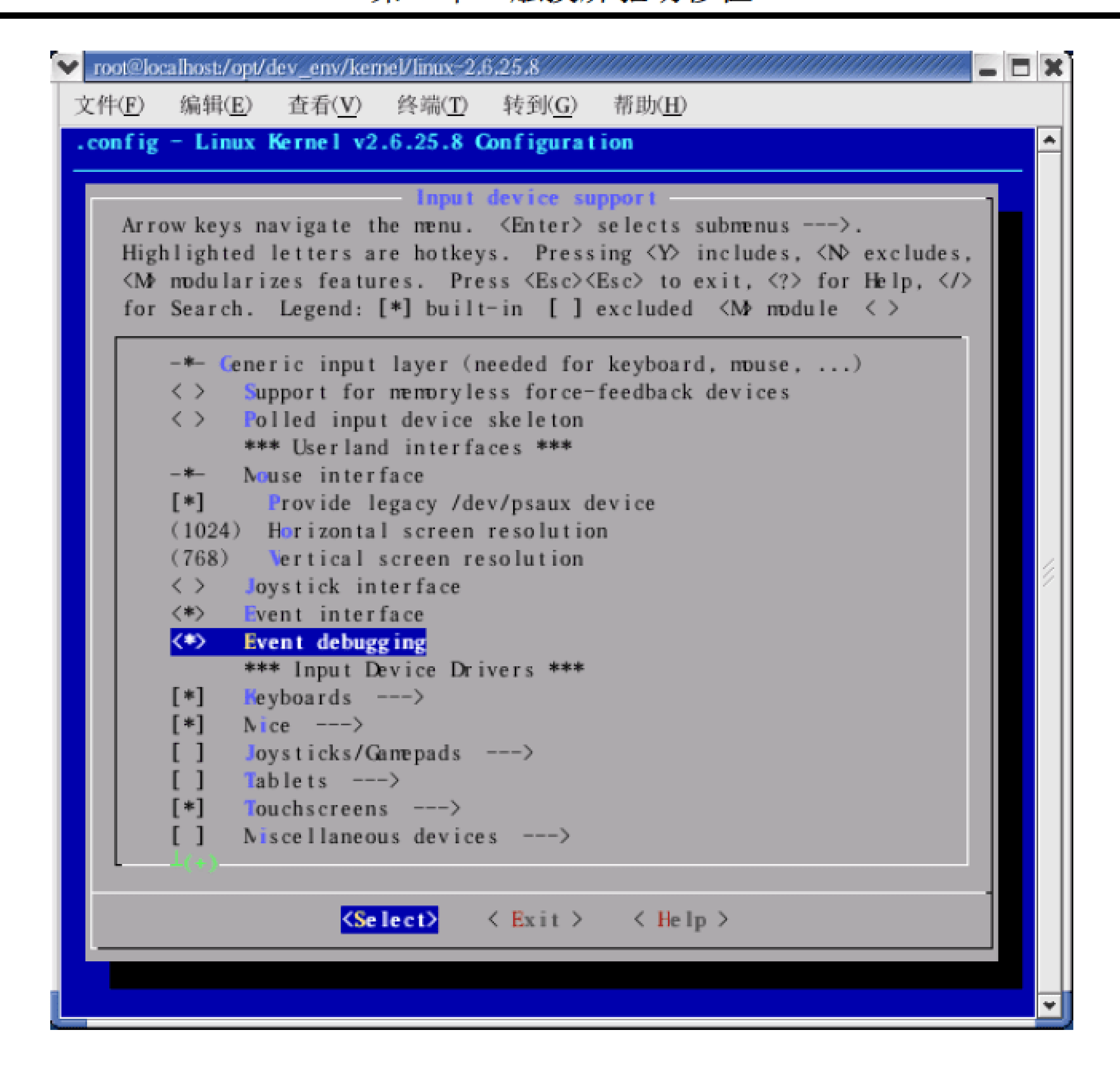



图 7.5　把触摸屏驱动编译进内核

## 7.5.5　触摸屏测试程序设计

对于输入事件接口的触摸屏设备，它使用的是输入设备的标准接口。这种接口传递的数据结构是 struct input_enent，它的定义在 include/linux/imput.h 中。

```
struct input_event {
struct timeval time;
unsigned short type;
unsigned short code;
unsigned int value;
};
```

其中，字段 time 是时间戳，利用结构体 timeval 记录了事件发生的时间；type 字段表明了事件的类型，对于触摸屏设备来说，在驱动程序中定义成了绝对输入设备（EV——ABS）；code 字段返回的是事件代码，在触摸屏设备中，只定义了 ABS_PRESSURE、ABS_X、ABS_Y；value 字段表示返回值，以下的代码可用于测试触摸屏驱动程序。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/tyupes.h>
#include <sys/ipc.h>
#include <sys/ioctl.h>
#include <pthread.h>
#include <fcntl.h>
#include <linux/input.h>

#define TS_DEV "/dev/inut/event0"
static int ts_fd=-1;
```

```
static int init_device(void)
{
    if((ts_fd=open(TS_DEV,O_RDONLY))<0){  //打开触摸屏设备
printf("open error:%s\n", TS_DEV);
return -1;
}
Return 0;
}

int main(void)
{
    int i;
    struct input_event data;
    if(init device()<0)
        return -1;

    for(;;){
        read(ts_fd,&data,siaeof(data));     //不停地读入数据
        if(data.type != EN_ABS)             //是否是绝对坐标设备
            printf("wrong type:%d\n",data,type);
        printf("event=%s,value=%d\n",data.code==ABS_X?"ABS_X":data.code
        ==ABS Y? "ABS Y": data.code==ABS PRESSURE? "ABS PRESSURE":"unkn-
        own",data.value);                   //打印数据
        }
}
```

# 7.6　小　　结

本章主要在分析了 S3C2440 ADC 控制寄存器的硬件操作，分析了内核 Input 子系统驱动源程序并在此基础上讲述了 s3c2440_ts.c 驱动程序的移植。触摸屏驱动程序移植主要是要了解内核中 Input 子系统体系结构及如何操作 LCD 控制器，因此本章的 7.2、7.4 节是后面移植工作的基础，读者要认真掌握。

# 第 8 章　USB 设备驱动移植

USB 是英文 Universal Serial Bus 的缩写，意为通用串行总线，是由 Compaq、DEC、IBM、Intel、NEC、Microsoft 及 Northern Telecom 等公司于 1994 年 11 月共同提出的，主要目的为了指定统一的 USB 标准。USB 设备使用起来比较方便，其文件传输速率快，而且具有热插拔性能，对于其应用越来越广泛，目前与 PC 连接的外围设备基本都具有 USB 接口。常见的 USB 设备包括 USB 鼠标、USB 键盘、USB 摄像头、USB 打印机、U 盘等。本章将主要结合代码详细介绍 USB 协议，并且讲解几种常见 USB 设备的移植步骤。

## 8.1　USB 协议

在 Linux 下进行 USB 设备驱动移植时，首先需要对 USB 协议有初步的了解。本节主要讲解 USB 协议的系统主要组成部分、USB 系统总线的拓扑结构、内部层次关系、数据流模式、USB 的调度等。

### 8.1.1　USB 协议的系统主要组成部分

USB 系统主要可分为 3 个部分：USB 的连接部分、USB 的设备和 USB 的主机。其分层模型如图 8.1 所示。

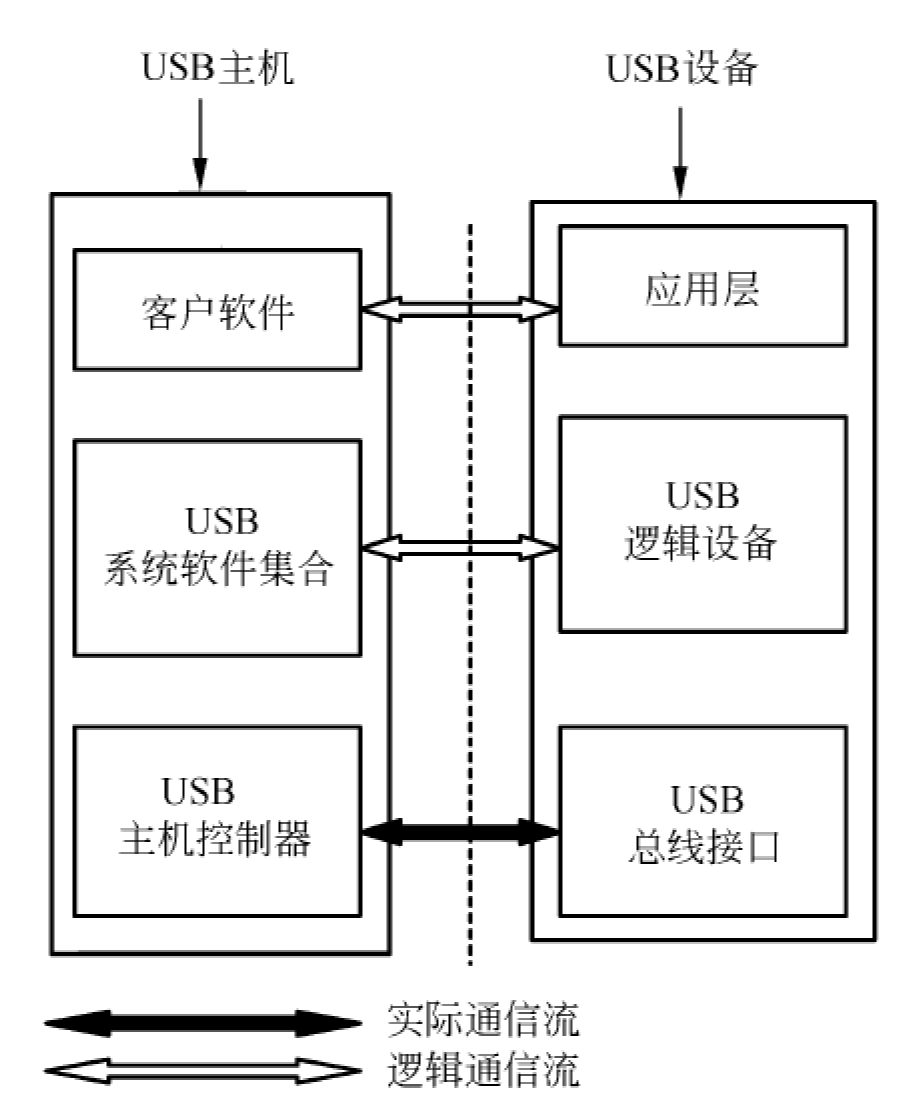



图 8.1　USB 主机、设备分层模型

USB 主机包括 USB 主机控制器、USB 系统软件集合、客户软件。其中，USB 系统软件集合由 USB 驱动程序、主机控制器的驱动程序和主机软件构成。USB 设备包括 USB 总线接口、USB 逻辑设备、应用层。

在 Linux 内核中，USB 设备采用结构体 usb_device 表示，该结构体的定义如下。

```
struct usb_device {
    int     devnum;                          //设备号，在 USB 总线上的地址
    char    devpath [16];                    //用于传递消息的 ID 字符串
    enum usb_device_state   state;
                            //设备状态：连接态、加电态、默认态、编址态、配置态
    enum usb device speed   speed;  //设备速度类型，全速、低速、高速其中一种
    struct usb_tt   *tt;                     //事务转换信息
    int     ttport;                          //事务转换 HUB 的设备端口
    unsigned int toggle[2];//每个端点用一个 bit 表示，[0]表示 IN，[1]表示 OUT
    struct usb_device *parent;  //USB 设备的父结点均为 HUB，根结点没有父结点
    struct usb_bus *bus;                     //USB 总线，USB 设备所在的总线
    struct usb_host_endpoint ep0;                //端点 0，默认的控制端点
    struct device dev;                           //通用设备接口
    struct usb_device_descriptor descriptor;     //USB 设备描述符
    struct usb_host_config *config;              //所有的设备配置
    struct usb_host_config *actconfig;           //活动的配置
    struct usb_host_endpoint *ep_in[16];         //从设备到主机的端点
    struct usb_host_endpoint *ep_out[16];        //从主机到设备的端点
    char **rawdescriptors;                       //每个配置的原始描述
    unsigned short bus_mA;                       //当前可用
    u8 portnum;                                  //父端口数
    u8 level;                                    //USB、HUB 祖先数
    unsigned can_submit:1;                       //可以被提交的 URB
    unsigned discon suspended:1;                 //挂起时断开
    unsigned persist_enabled:1;                  //该 USB 设备使能 USB_PERSIST
    unsigned have langid:1;                      //tring langid 是否有效
    unsigned authorized:1;                       //权限
    unsigned authenticated:1;                    //通过鉴权
    unsigned wusb:1;                             //无线 USB 设备
    int string_langid;                           //字符串语言标识
    /*下面是设备的固有属性，包括产品 ID，生产字符串，生产序列号等*/
    char *product;
    char *manufacturer;
    char *serial;
    struct list_head filelist;
#ifdef CONFIG USB DEVICE CLASS
    struct device *usb_classdev;                 //USB 类设备
#endif
#ifdef CONFIG_USB_DEVICEFS
    struct dentry *usbfs_dentry;
#endif
    int maxchild;                  //最多能接的子设备个数，也就是 HUB 的端口数
    struct usb_device *children[USB_MAXCHILDREN];  //接在 HUB 上的子设备
    int pm_usage_cnt;                              //使用计数
    u32 quirks;
    atomic t urbnum;                               //提交的 URB 数目
    unsigned long active duration;
```

```
#ifdef CONFIG_PM
    struct delayed_work autosuspend;
    struct work_struct autoresume;
    struct mutex pm_mutex;

    unsigned long last_busy;
    int autosuspend_delay;
    unsigned long connect_time;                 //设备第一次连接时间

    unsigned auto_pm:1;
    unsigned do_remote_wakeup:1;                //使能远程唤醒
    unsigned reset_resume:1;                    //重设而不是重启
    unsigned autosuspend_disabled:1;            //用户禁止自动挂起
    unsigned autoresume_disabled:1;             //用户禁止自动重启
    unsigned skip_sys_resume:1;                 //跳过下一个系统重启
#endif
    struct wusb_dev *wusb_dev;      //如果是无线设备，则为设备连接 WUSB 专门数据
};
```

## 8.1.2　总线物理拓扑结构

USB 系统中的主机和设备采用的是星形连接方式，其物理连接拓扑图如图 8.2 所示。

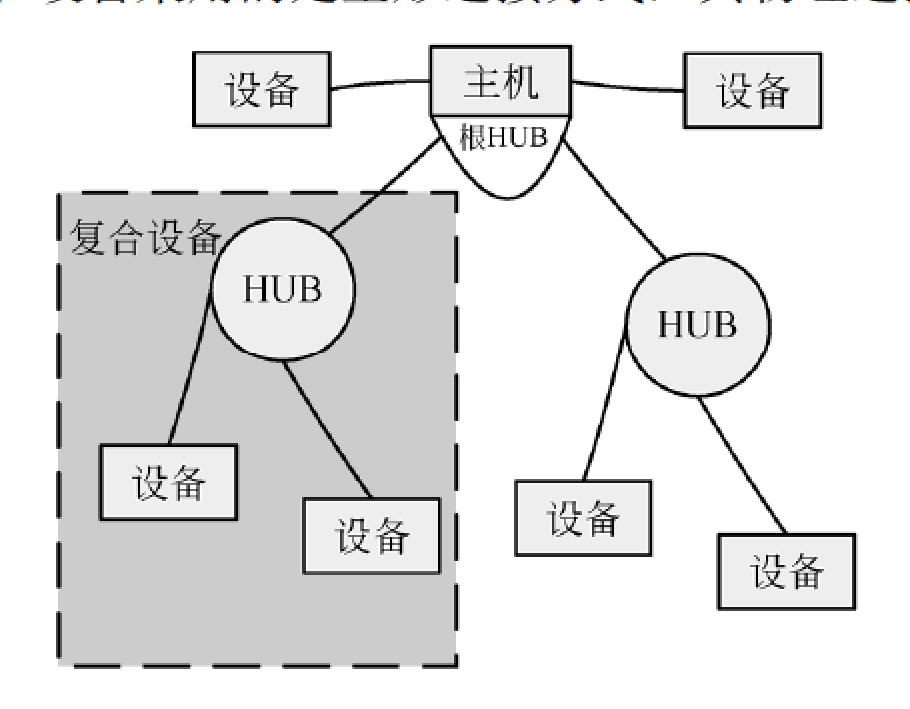



图 8.2　USB 物理总线的拓扑

图 8.2 中的 HUB 是一类特殊的 USB 设备，它是一组 USB 的连接点，主机中有一个被嵌入的 HUB 叫根 HUB（root Hub），主机通过根 HUB 提供多个连接点。为了防止出现环状连接，连接点采用星形连接来体现层次性。

## 8.1.3　USB 设备、配置、接口、端点

在 USB 设备的逻辑组织分层结构中，包含设备、配置、接口和端点 4 个层次。

每个 USB 设备都提供了不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个）；每个配置则由多个接口组成；接口由多个端点组成，每个接口代表一个基本的功能，是 USB 设备驱动程序控制的对象，一个复杂的 USB 设备可以具有多个接口。

端点是 USB 通信的最基本形式，对主机来说，每一个 USB 设备接口就是一组端点的集合。主机只有通过端点才能和设备进行通信，以使用设备的功能。在 USB 系统中每个端

点都有独一无二的地址，该地址由设备地址和端点号指定。每个端点都有一组属性，其中包括传输方式、总线访问频率、带宽、端点号和数据包的最大容量等。一个 USB 端点只能在一个方向承载数据，或者从主机到设备（称为输出端点），或者从设备到主机（称为输入端点），因此端点传输是单向的。端点 0 通常为控制端点，用于初始化设备参数。只要设备连接到 USB 上并且上电，端点 0 就可以被访问。端点 1、2 等一般用作数据端点，存放主机与设备间通信的数据。

USB 设备非常复杂，由许多不同的逻辑单元组成，如图 8.3 所示。

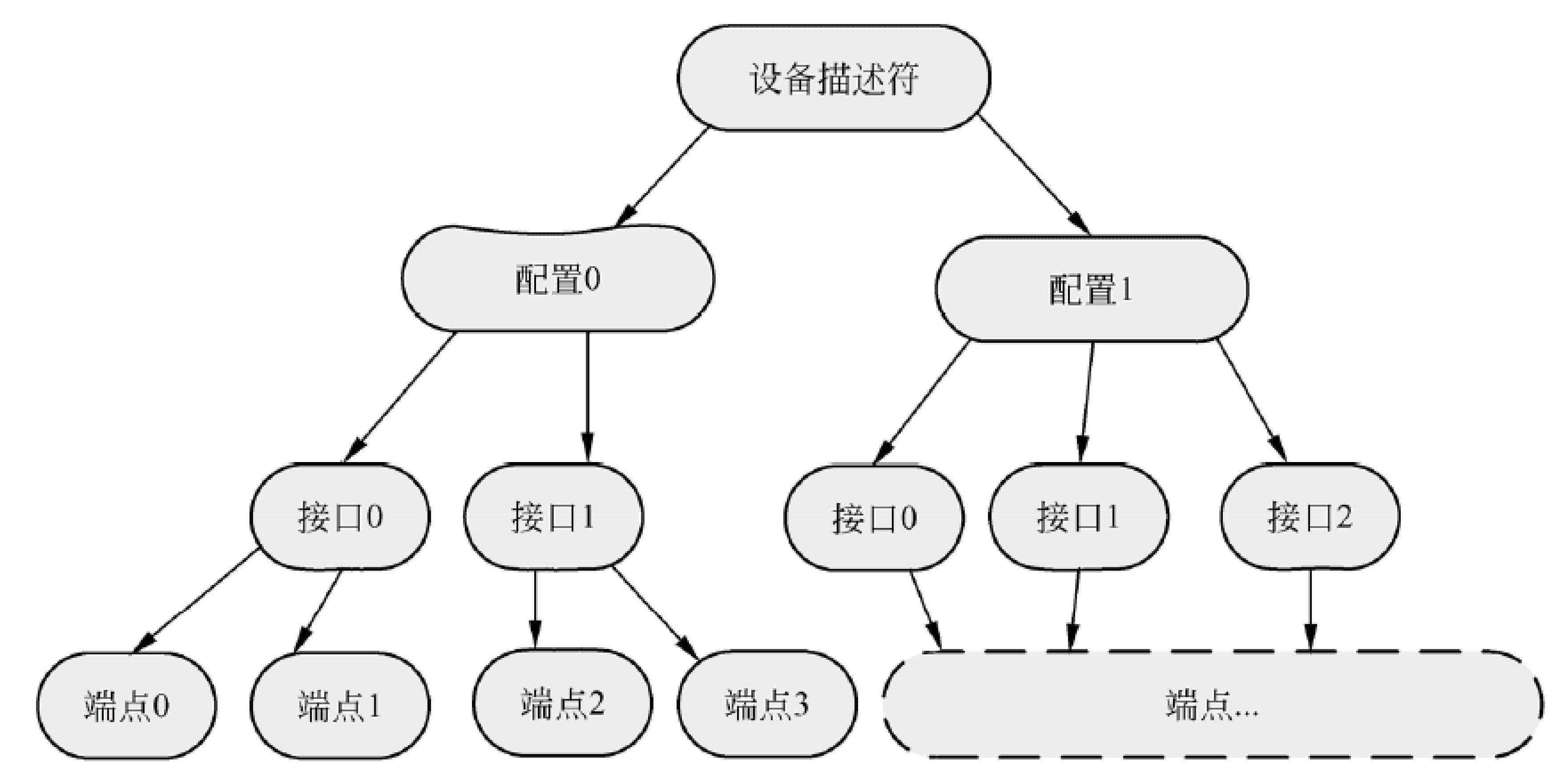



图 8.3　USB 设备、配置、接口和端点

- 设备通常有一个或多个配置；
- 配置通常有一个或多个接口；
- 接口通常有一个或多个设置；
- 接口有 0 个或多个端点。

下面分别结合代码来介绍设备描述符、配置描述符、接口描述符、端点描述符。

### 1．设备描述符

设备描述符是关于设备的通用信息，包括供应商 ID、产品 ID 和修订 ID、支持的设备类、子类和适用的协议以及默认端点的最大包大小等。在 Linux 内核中，USB 设备用 usb_device 结构体来描述，USB 设备描述符被定义为 usb_device_descriptor 结构体，该结构体的定义代码如下：

```
struct usb_device_descriptor {
    __u8  bLength;              //描述符长度
    __u8  bDescriptorType;      //描述符类型编号

    __le16 bcdUSB;              //USB 版本号
    __u8  bDeviceClass;         //USB 分配的设备类 code
    __u8  bDeviceSubClass;      //USB 分配的子类 code
    __u8  bDeviceProtocol;      //USB 分配的协议 code
    __u8  bMaxPacketSize0;      //endpoint0 最大包大小
    __le16 idVendor;            //厂商编号
    __le16 idProduct;           //产品编号
```

```
    __le16 bcdDevice;              //设备出厂编号
    __u8  iManufacturer;           //描述厂商字符串的索引
    __u8  iProduct;                //描述产品字符串的索引
    __u8  iSerialNumber;           //描述设备序列号字符串的索引
    __u8  bNumConfigurations;      //可能的配置数量
}    attribute    ((packed));
```

### 2. 配置描述符

配置描述符中包括该配置中的接口数、支持的挂起和恢复能力以及功率要求。USB 配置描述符在内核中被定义为 usb_host_config 结构体，结构体 usb_config_descriptor 的定义代码如下：

```
struct usb config descriptor {
    __u8  bLength;                 //描述符长度
    __u8  bDescriptorType;         //描述符类型编号

    __le16 wTotalLength;           //配置所返回的所有数据的大小
    __u8  bNumInterfaces;          //配置所支持的接口数
    __u8  bConfigurationValue;     //Set_Configuration 命令需要的参数值
    __u8  iConfiguration;          //描述该配置字符串的索引值
    __u8  bmAttributes;            //供电模式的选择
    __u8  bMaxPower;               //设备从总线提取的最大电流
}    attribute    ((packed));
```

### 3. 接口描述符

接口描述符中包括接口类、子类和适用的协议，接口备用配置的数目和端点数目。USB 接口描述符在内核中被定义为 usb_interface 结构体，结构体 usb_interface_descriptor 的定义代码如下：

```
struct usb_interface_descriptor {
    __u8  bLength;                 //描述符长度
    __u8  bDescriptorType;         //描述符类型
    __u8  bInterfaceNumber;        //接口的编号
    __u8  bAlternateSetting;       //备用的接口描述符编号
    __u8  bNumEndpoints;           //该接口使用的端点数，不包括端点 0
    __u8  bInterfaceClass;         //接口类型
    __u8  bInterfaceSubClass;      //接口子类型
    __u8  bInterfaceProtocol;      //接口所遵循的协议
    __u8  iInterface;              //描述该接口的字符串索引值
}    attribute    ((packed));
```

### 4. 端点描述符

端点描述符中包括端点地址、方向和类型，支持的最大包大小。如果端点为中断类型，则端点描述符中还包括轮询频率。在 Linux 内核中，USB 端点被定义为 usb_host_endpoint 结构体，usb_endpoint_descriptor 结构体的代码定义如下：

```
struct usb endpoint descriptor {
```

```
    __u8  bLength;            //描述符长度
    __u8  bDescriptorType;    //描述符类型

    __u8  bEndpointAddress;
                              //端点地址：0～3 位是端点号，第 7 位是方向(0-OUT,1-IN)
    /*端点属性：bit[0:1] 的值为 00 表示控制，为 01 表示同步，为 02 表示批量，为 03 表示
    中断*/
    __u8  bmAttributes;
    __le16 wMaxPacketSize;    //本端点接收或发送的最大信息包的大小
    __u8  bInterval;          //轮询数据传送端点的时间间隔
                              //对于批量传送的端点及控制传送的端点，此域忽略
                              //对于同步传送的端点，此域必须为 1
                              //对于中断传送的端点，此域值的范围为 1～255
    /*注意：下面两个字段仅在音频端点中使用*/
    __u8  bRefresh;
    __u8  bSynchAddress;
} __attribute__((packed));
```

#### 5. 字符串描述符

字符串描述符的功能是在其他描述符中为某些字段提供字符串索引，用来检索描述性字符串，可以采用多种语言形式提供。字符串描述符是可选的，有些设备具有该描述符，而另外一些设备则可能没有该描述符，字符串描述符被定义为 usb_string_descriptor 结构体，usb_string_descriptor 结构体的定义代码如下：

```
struct usb_string_descriptor {
    __u8  bLength;                //描述符长度
    __u8  bDescriptorType;        //描述符类型
    __le16 wData[1];              //UTF-16LE 编码
} __attribute__ ((packed));
```

### 8.1.4　USB 设备状态

USB 设备状态分为 6 种状态，分别是连接态、加电态、默认态、编址态、配置态和挂起态。各个状态之间的状态转移关系如图 8.4 所示。

- ❑ 加电态：USB 设备的电源可来自外部电源，也可从 USB 接口的集线器而来。电源来自外部电源的 USB 设备被称作自给电源式的（self-powered）USB 设备。尽管自给电源式的 USB 设备在连接上 USB 接口前可能已经处于带电状态，但它们连接到 USB 接口后才能被看作是加电状态（Powered state）。
- ❑ 默认态：设备加电以后，在从总线接收到复位信号前不应对总线传输发生响应。只有设备在接收到复位信号之后，才能在默认地址处变为可寻址。
- ❑ 编址态：在加电复位后所有的 USB 设备都使用默认地址与主机通信。每个设备在连接或复位后由主机分配一个唯一的地址。当 USB 设备被挂起时，它保持这个地址不变。

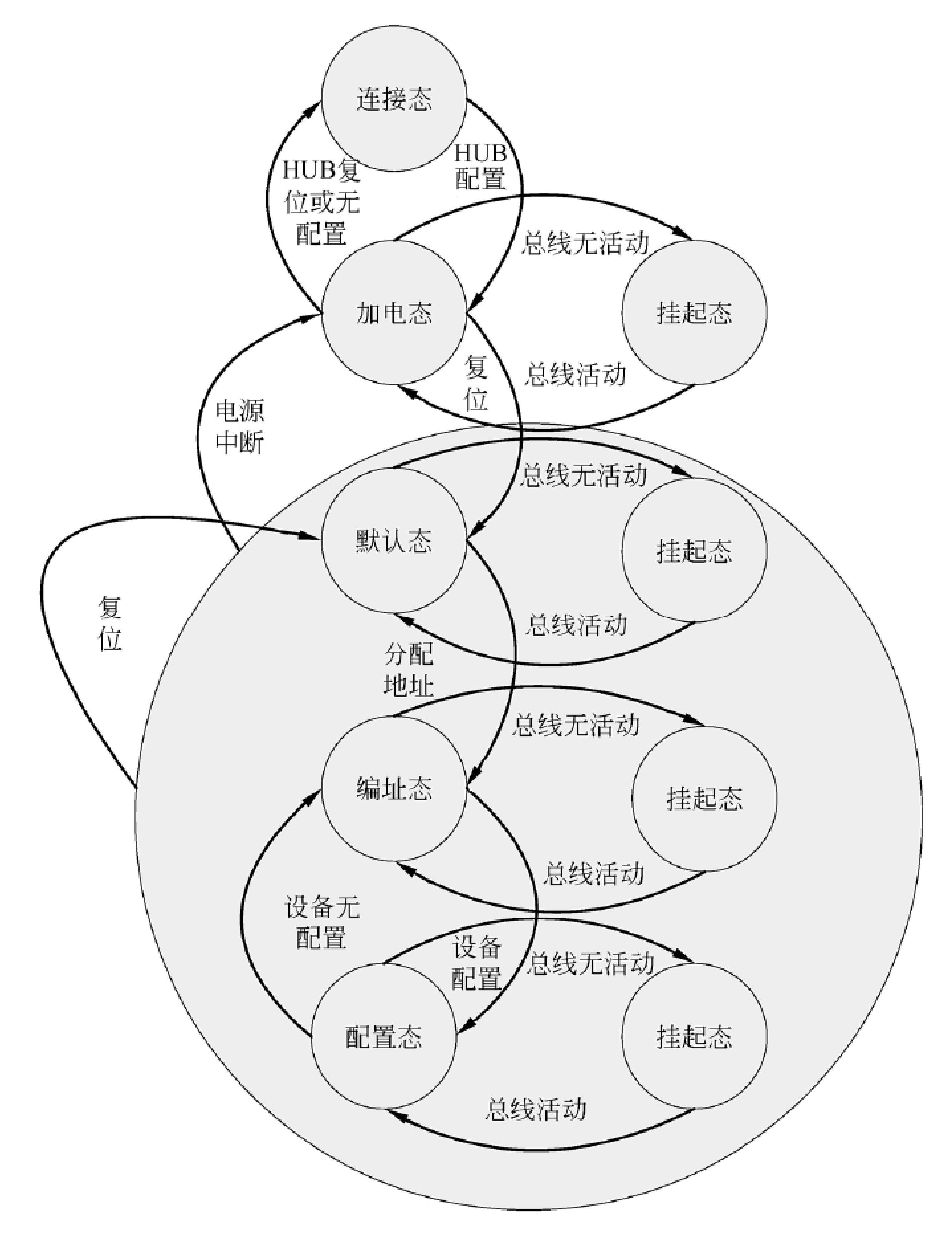



图 8.4　USB 设备状态转化图

- 配置态：在 USB 设备正常工作以前，设备必须被正确配置。从设备的角度来看，配置包括将非 0 值写入设备配置寄存器的操作。配置一个设备或改变一个可变的设备设置会使得与这个相关接口的终端结点的所有状态与配置值被设成默认值。
- 挂起态：为了省电，USB 设备在探测不到总线传输时自动进入中止状态。当中止时，USB 设备保持本身的内部状态不变，包括它的地址及配置。

## 8.1.5　USB 枚举过程

了解 USB 设备状态后，进一步结合代码和序列图了解 USB 的枚举过程。USB 的枚举过程如图 8.5 所示。主机集线器监视每个端口的信号电压，当有新设备接入时就能被检测到。当 USB 设备插入到 HUB 后，便开始 USB 的枚举过程。USB 的枚举过程主要分为以下几个步骤：

（1）Get_Port_Status：主机发现最新接入的设备，返回消息告诉主机新接入的设备是何时连接到集线器上的。

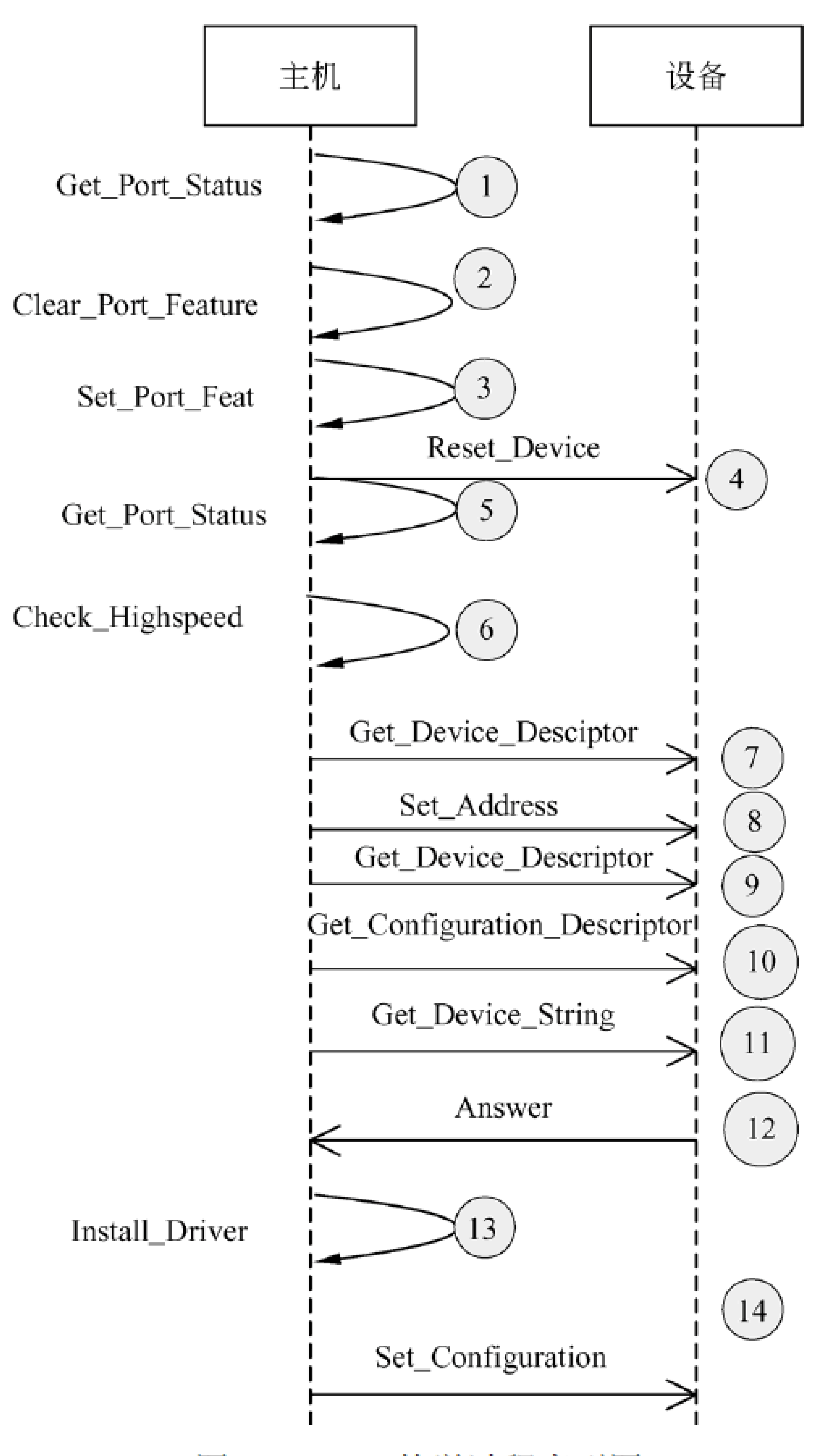



图 8.5　USB 枚举过程程序列图

```
static int get_port_status(struct usb_device *hdev, int port1, struct
usb_port_status *data)
{
    int i, status = -ETIMEDOUT;
    for (i = 0; i < USB STS RETRIES && status == -ETIMEDOUT; i++) {
        /*从设备 hdev 的端点 0 请求 USB 设备状态，将读取的设备数据保存在 data*/
        status = usb control msg(hdev, usb rcvctrlpipe(hdev, 0),
            USB_REQ_GET_STATUS, USB_DIR_IN | USB_RT_PORT, 0, port1,
            data, sizeof(*data), USB_STS_TIMEOUT);
    }
    return status;
}
```

（2）Clear_Port_Feature：用于清除 STATUS__CHANGE 寄存器中的标志。

```
static int clear_port_feature(struct usb_device *hdev, int port1, int
feature)
{
    /*清除设备 hdev 端口信息*/
    return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
        USB_REQ_CLEAR_FEATURE, USB_RT_PORT, feature, port1,
        NULL, 0, 1000);
}
```

（3）Set_Port_Feature：当主机知道有新的设备时，主机向集线器发送 Set_Port_Feature 请求，请求集线器重新设置端口。集线器使得设备的 USB 数据线处于重启（RESET）状态至少 10ms。

```
static int set_port_feature(struct usb_device *hdev, int port1, int feature)
{
    /*设置设备 hdev 端口信息*/
    return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
        USB_REQ_SET_FEATURE, USB_RT_PORT, feature, port1,
        NULL, 0, 1000);
}
```

（4）Reset_Device：重启设备。

```
int usb_reset_device(struct usb_device *udev);
```

（5）Get_Port_Status：主机发送一个 Get_Port_Status 请求验证设备是否激起重启状态。返回的数据有一位表示设备仍然处于重启状态。集线器释放重启状态后，设备就进入了默认状态，即设备已准备好通过 Endpoint 0 的默认流程响应控制传输。即设备目前使用默认地址 0x0 和主机通信。

（6）集线器检测设备速度：集线器通过测定那根信号线（D+或 D–）在空闲时有更高的电压来检测设备是低速设备还是全速设备。然后通过 check_highspeed()检测是全速设备还是高速设备。

```
static void check_highspeed (struct usb_hub *hub, struct usb_device *udev,
int port1)
{
    struct usb_qualifier_descriptor    *qual;
    int                status;
    /*给结构体 usb_qualifier_descriptor 分配空间*/
    qual = kmalloc (sizeof *qual, GFP_KERNEL);
    if (qual == NULL)
        return;
    /*从设备 udev 读取设备 USB 描述符，该描述的类型为 USB_DT_DEVICE_QUALIFIER*/
    status = usb_get_descriptor (udev, USB_DT_DEVICE_QUALIFIER, 0, qual,
    sizeof *qual);
    if (status == sizeof *qual) {
        dev_info(&udev->dev, "not running at top speed; "
            "connect to a high speed hub\n");
        /*hub LEDs are probably harder to miss than syslog*/
        if (hub->has_indicators) {
            hub->indicator[port1-1] = INDICATOR_GREEN_BLINK;
            /*加入工作者线程 hub->leds，等待时间为 0 后执行*/
            schedule_delayed_work (&hub->leds, 0);
        }
    }
    /*释放所分配结构体 qual 的空间*/
    kfree(qual);
}
```

（7）Get_Device_Desciptor：以取得默认控制管道所支持的最大数据包长度，并在有限的时间内等待 USB 设备的响应，该长度包含在设备描述符的 bMaxPacketSize0 字段中，其地址偏移量为 7，所以这时主机只需读取该描述符的前 8 个字节。

（8）Set_Address：PC 主机分配一个设备地址给新接入的 I/O 设备，所有随后的请求都

将会发送到这个新的设备地址。接收新分配的设备地址后，设备进入到已编址状态。

```
static int hub_set_address(struct usb_device *udev, int devnum)
{
    int retval;
    if (devnum <= 1)
        return -EINVAL;
    if (udev->state == USB_STATE_ADDRESS)
        return 0;
    if (udev->state != USB_STATE_DEFAULT)
        return -EINVAL;
    /*向设备 udev 发送请求设置地址*/
    retval = usb_control_msg(udev, usb_sndaddr0pipe(),
        USB_REQ_SET_ADDRESS, 0, devnum, 0,
        NULL, 0, USB_CTRL_SET_TIMEOUT);
    if (retval == 0) {
        /*更新设备的新地址*/
        update_address(udev, devnum);
        /*设置 USB 设备状态为 USB_STATE_ADDRESS*/
        usb_set_device_state(udev, USB_STATE_ADDRESS);
        /*重新初始化设备 udev 的端点 0*/
        usb_ep0_reinit(udev);
    }
    return retval;
}
```

（9）Get_Device_Descriptor：读取设备描述符的全部字段，获得该设备的总体信息，包括 VID、PID 等。

（10）Get_Configuration_Descriptor：设备驱动程序开始读取所有关于设备的信息，包括设备的接口和端点。对于一个功能复杂的 I/O 设备，配置可能相当庞大。如果设备有多个配置，驱动程序也要读出所有配置。

（11）Get_Device_String：主机发送 Get_Device_String 命令给设备，获得字符集描述（unicode），比如厂商、产品描述、型号等。

（12）根据 Device_Descriptor 和 Device_Configuration 应答。将设备、配置、端点描述符等信息反馈给主机，方便主机正确加载设备驱动程序。

（13）安装设备驱动程序：PC 主机需要决定用哪个设备驱动程序去支持新连接的 USB 设备。如果选定的设备驱动程序没有加载到内存中，就要立即加载设备驱动程序到内存中。

（14）Set_Configuration：现在设备已经被配置好了并且可以运行，此时设备进入已配置状态。

### 8.1.6　USB 请求块（URB）

USB 请求块（USB request block，URB）是 USB 设备驱动中用来描述与 USB 设备通信所用的基本载体和核心数据结构，与网络设备驱动中的 sk_buff 结构体类似，是 USB 主机与设备之间传输数据的封装。下面是对 URB 结构体的定义。

```
struct urb {
    /*私有的：只能由 USB 核心和主机控制器访问的字段*/
    struct kref kref;                   //URB 引用计数
    void *hcpriv;                       //主机控制器的私有数据
```

```
    atomic_t use_count;                     //并发传输计数
    atomic_t reject;                        //传输将失败
    int unlinked;                           //未连接错误码
    /*公共的：可以被驱动使用的字段*/
    struct list_head urb_list;              //当前使用的 URB 链表头
    struct list_head anchor_list;           //the URB may be anchored
    struct usb_anchor *anchor;
    struct usb_device *dev;                 //关联的 USB 设备
    struct usb_host_endpoint *ep;           //端点指针
    unsigned int pipe;                      //管道信息
    int status;                             //URB 的当前状态
    unsigned int transfer_flags;            //(in) URB_SHORT_NOT_OK | ...
    void *transfer_buffer;                 //发送数据到设备或从设备接收数据的缓冲区
    dma_addr_t transfer_dma;                //用来以 DMA 方式向设备传输数据的缓冲区
    int transfer_buffer_length;
                        //transfer_buffer 或 transfer_dma 指向缓冲区的大小
    int actual_length;                      //URB 结束后，发送或接收数据的实际长度
    unsigned char *setup_packet;            //指向控制 URB 设置数据包的指针
    dma_addr_t setup_dma;                   //控制 URB 设置数据包的 DMA 缓冲区
    int start_frame;                        //等时传输中用于设置或返回初始帧
    int number_of_packets;                  //等时传输中包数
    int interval;           //URB 被轮询到的时间间隔（对中断和等时 urb 有效）
    int error_count;                        //等时传输错误数量
    void *context;                          //completion()函数上下文
    usb_complete_t complete;    //当 URB 被完全传输或发生错误时，被调用
    struct usb_iso_packet_descriptor iso_frame_desc[0];
                /*单个 URB 一次可定义多个等时传输时，描述各个等时传输*/
};
```

当 transfer_flags 标志中的 URB_NO_TRANSFER_DMA_MAP 被设置时，USB 核心将使用 transfer_dma 指向的缓冲区而不使用 transfer_buffer 指向的缓冲区，这表示即将传输 DMA 缓冲区。

当 transfer_flags 标志中的 URB_NO_SETUP_DMA_MAP 被设置时，如果控制 urb 有 DMA 缓冲区，USB 核心将使用 setup_dma 指向的缓冲区而不使用 setup_packet 指向的缓冲区。

URB 是 USB 接口通信的关键数据，下面介绍 URB 处理流程的几个重要函数。

### 1．URB创建函数

创建 URB 的过程主要包括为 URB 分配空间，并初始化该 URB 结构体。

```
struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
{
    struct urb *urb;
    /*为 URB 分配空间*/
    urb = kmalloc(sizeof(struct urb) +
        iso_packets * sizeof(struct usb_iso_packet_descriptor),
        mem_flags);
    if (!urb) {
        printk(KERN_ERR "alloc_urb: kmalloc failed\n");
        return NULL;
    }
    /*初始化该 URB，后面将会介绍初始化 URB 的具体实现*/
```

```
    usb_init_urb(urb);
    return urb;
}
```

参数 iso_packets 表示这个 URB 应当包含的等时数据包的个数，若为 0 表示不创建等时数据包。参数 mem_flags 为分配内存的标志。如果分配成功，该函数返回一个 URB 结构体指针，分配失败则返回 0。在驱动中不要静态创建 URB 结构体，否则可能破坏 USB 核心给 URB 使用的引用计数方法。

### 2．初始化URB函数

根据 USB 的设备端点类型来分，有 3 种初始化函数。

（1）对于中断 URB，其初始化函数为：

```
static inline void usb_fill_int_urb(struct urb *urb,
                   struct usb_device *dev,
                   unsigned int pipe,
                   void *transfer_buffer,
                   int buffer_length,
                   usb complete t complete fn,
                   void *context,
                   int interval)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer buffer = transfer buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
    if (dev->speed == USB_SPEED_HIGH)
        urb->interval = 1 << (interval - 1);
    else
        urb->interval = interval;
    urb->start_frame = -1;
}
```

参数 urb 指向要被初始化的 URB 的指针；参数 dev 指向这个 URB 要被发送到的 USB 设备；参数 pipe 是这个 URB 要被发送到的 USB 设备的特定端点；参数 transfer_buffer 是指向发送数据或接收数据的缓冲区的指针，和 URB 一样，它也不能是静态缓冲区，必须使用 kmalloc()来分配；参数 buffer_length 是 transfer_buffer 指针所指向缓冲区的大小；complete 指针指向当这个 URB 完成时被调用的完成处理函数；参数 context 是完成处理函数的“上下文”；参数 interval 是这个 URB 应当被调度的间隔。

（2）对于控制 URB，其初始化函数为：

```
static inline void usb_fill_control_urb(struct urb *urb,
                   struct usb_device *dev,
                   unsigned int pipe,
                   unsigned char *setup_packet,
                   void *transfer_buffer,
                   int buffer length,
                   usb complete t complete fn,
                   void *context)
{
```

```
    urb->dev = dev;
    urb->pipe = pipe;
    urb->setup packet = setup packet;
    urb->transfer buffer = transfer buffer;
    urb->transfer buffer length = buffer length;
    urb->complete = complete_fn;
    urb->context = context;
}
```

（3）对于批量 URB，其初始化函数为：

```
static inline void usb_fill_int_urb(struct urb *urb,
                   struct usb_device *dev,
                   unsigned int pipe,
                   void *transfer_buffer,
                   int buffer length,
                   usb complete t complete fn,
                   void *context,
                   int interval)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer buffer = transfer buffer;
    urb->transfer buffer length = buffer length;
    urb->complete = complete fn;
    urb->context = context;
    if (dev->speed == USB_SPEED_HIGH)
        urb->interval = 1 << (interval - 1);
    else
        urb->interval = interval;
    urb->start_frame = -1;
}
```

批量 URB 和控制 URB 与中断 URB 的参数基本类似。

### 3．释放URB

释放由 usb_alloc_urb()分配的 URB 结构体，释放 URB 的函数如下：

```
void usb_free_urb(struct urb *urb)
{
    if (urb)
        kref_put(&urb->kref, urb_destroy);
}
```

### 4．提交URB

提交 URB 给 USB 核心。在完成分配并设置 URB 后，使用 usb_submit_urb()函数把新的 URB 提交到 USB 核心，提交 URB 函数定义如下。

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
```

参数 URB 指向被提交的 URB 结构体，参数 mem_flags 是传递给 USB 核心的内存选

项，该参数用于指示 USB 核心如何分配内存缓冲区。如果函数执行成功，URB 的控制权将被 USB 核心接管，否则函数返回错误。

# 8.2　USB 主机驱动

为了达到 USB 主机功能统一、提高系统的可靠性与可移植性的目的，芯片厂家同时确定 USB 标准和相应的主机规范。现在用得比较广泛的有 3 种：INTEL 推出的用于 USB 2.0 高速设备的 EHCI （Enhanced Host Control Interface 增强主机控制接口）规范和 UHCI（Universal Host Control Interface 通用主机）规范，以及前 Compaq 、Microsoft 等推出的可用于全速与低速 USB 系统中的 OHCI （Open Host Control Interface 开放主机控制接口）规范。

## 8.2.1　USB 主机驱动结构和功能

USB 主机控制器有 3 种标准：OHCI、UHCI 和 EHCI。OHCI 对硬件的要求与系统性能、软件复杂的要求相对较低，也能够满足大部分具有 USB 接口嵌入式系统的要求。UHCI 对系统的处理能力与软件的开发要求相对要高（PC 就较多地采用了 UHCI）；EHCI 兼容 OHCI 和 UHCI。在嵌入式的 USB HOST 功能中，采用 OHCI 的规范，后面将主要介绍 OHCI 主机驱动。

Linux 中的 USB 子系统核心模块为 USB Core 模块，它为 USB 驱动（device 和 HC）提供了一个用于访问和控制 USB 硬件的统一接口。应用程序发送的 USB 请求块（urb）经过 USB 设备驱动和 USB Core 后到达 USB 主机控制器（HC），主机控制器解析 URB 后将数据发往指定的 USB 设备，如图 8.6 所示。

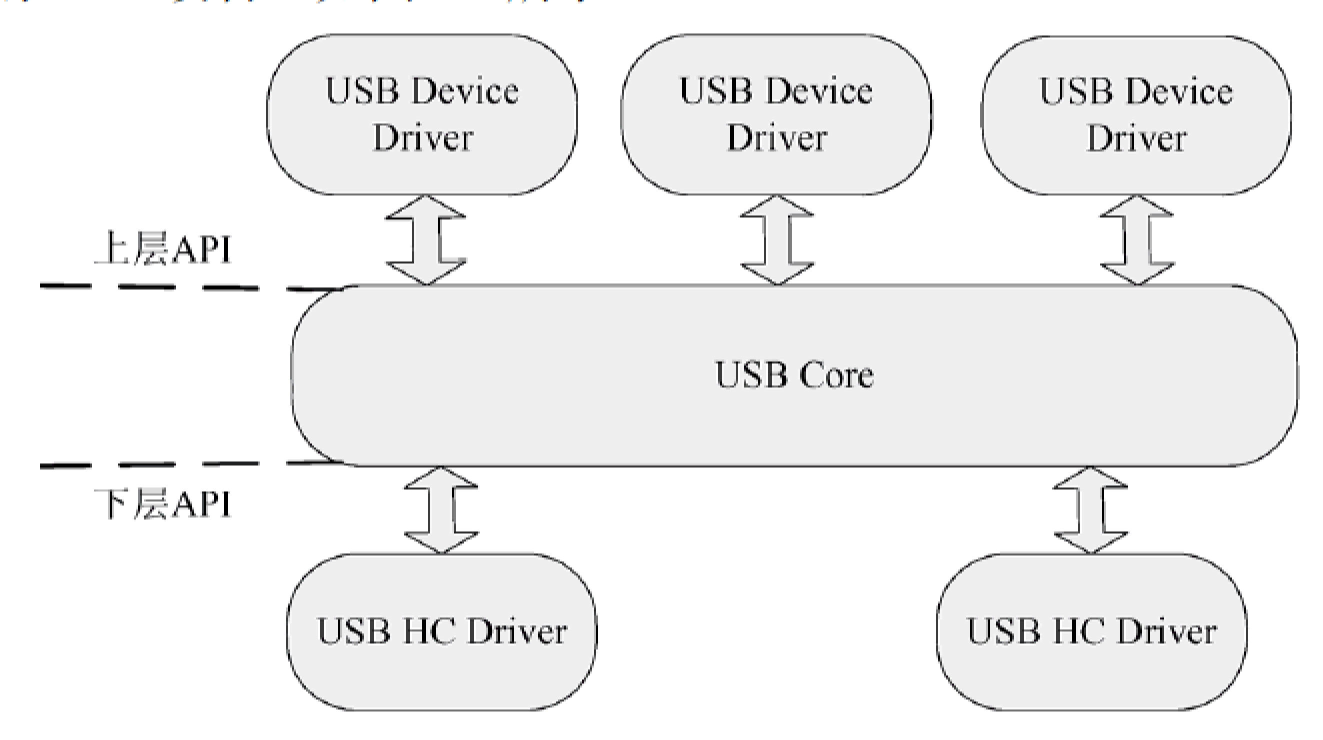



图 8.6　USB 驱动结构

主机控制器驱动程序完成的功能主要包括以下几点：

- 解析和维护 URB，根据不同的端点进行分类缓存 URB。
- 负责不同 USB 传输类型的调度工作。
- 负责 USB 数据的实际传输工作。
- 实现虚拟根 HUB 的功能。

### 8.2.2　主机控制器驱动（usb_hcd）

Linux 内核中，USB 主机控制器驱动被定义为 usb_hcd 结构体。在 usb_hcd 结构体中描述了 USB 主机控制器的硬件信息、状态和操作函数。usb_hcd 的定义在内核 drivers/usb/core/hcd.h 文件中，其定义代码如下：

```
struct usb_hcd {
    /*控制器的基本信息*/
    struct usb_bus      self;           //hcd is-a bus
    struct kref     kref;               //索引计数器
    const char      *product_desc;      //产品/厂商字符串
    char            irq_descr[24];      //驱动+总线#
    struct timer_list   rh_timer;       //根 HUB 轮询时间间隔
    struct urb      *status_urb;        //当前 URB 状态
#ifdef CONFIG_PM
    struct work_struct  wakeup_work;    /*针对具备唤醒能力的 USB 设备，USB 设备也可以远程唤醒的电流信号来请求主机退出中止态或选择中止态。当设备复位时，远程唤醒能力必须被禁止。*/
#endif
    /*硬件信息和状态*/
    const struct hc_driver *driver;     //控制器驱动使用的回调函数

    /*需要维护的标志*/
    unsigned long       flags;
#define HCD_FLAG_HW_ACCESSIBLE 0x00000001
#define HCD_FLAG_SAW_IRQ    0x00000002

    unsigned        rh_registered:1;    //是否注册根 HUB
    /*The next flag is a stopgap, to be removed when all the HCDs
     * support the new root-hub polling mechanism.*/
    unsigned        uses_new_polling:1;
    unsigned        poll_rh:1;          //是否允许轮询根 HUB 状态
    unsigned        poll_pending:1;     //状态是否改变
    unsigned        wireless:1;         //是否支持无线 USB 主机控制器
    unsigned        authorized_default:1;
    unsigned        has_tt:1;           //Integrated TT in root hub

    int             irq;                //控制器的中断请求号
    void __iomem        *regs;          //控制器使用的内存和 I/O
    u64             rsrc_start;         //控制器使用的内存和 I/O 起始地址
    u64             rsrc_len;           //控制器使用的内存和 I/O 资源长度
    unsigned        power_budget;       //in mA, 0 = 无限制

#define HCD_BUFFER_POOLS    4
    struct dma_pool     *pool [HCD_BUFFER_POOLS];
    int             state;
#   define  __ACTIVE        0x01
#   define  __SUSPEND       0x04
#   define  __TRANSIENT     0x80
#   define  HC_STATE_HALT       0
#   define  HC_STATE_RUNNING    (__ACTIVE)
#   define  HC_STATE_QUIESCING  (__SUSPEND|__TRANSIENT|__ACTIVE)
#   define  HC_STATE_RESUMING   (__SUSPEND|__TRANSIENT)
```

```
#   define  HC_STATE_SUSPENDED  (__SUSPEND)
#define HC_IS_RUNNING(state) ((state) & __ACTIVE)
#define HC_IS_SUSPENDED(state) ((state) &  SUSPEND)
    /*主机控制器的私有数据*/
    unsigned  long  hcd priv[0]    attribute    ((aligned(sizeof(unsigned
long))));
};
```

usbs_hcd 结构体中的 hc_driver 成员也比较重要，该结构包含了具体的用于操作主机控制器的回调函数。该结构体的定义如下：

```
struct hc driver {
    const char*description;                  //"ehci-hcd" 等
    const char*product_desc;                 //产品/厂商字符串
    size_t      hcd_priv_size;               //私有数据的大小*/
    /*中断处理函数*/
    irqreturn_t (*irq) (struct usb_hcd *hcd);
    int flags;
#define HCD_MEMORY  0x0001                   //HC 寄存器使用的内存和 I/O
#define HCD_LOCAL_MEM   0x0002               //HC 需要本地内存
#define HCD_USB11   0x0010                   //USB 1.1 协议
#define HCD_USB2    0x0020                   //USB 2.0 协议

    /*用来初始化 HCD 和 root hub*/
    int (*reset) (struct usb_hcd *hcd);
    int (*start) (struct usb_hcd *hcd);
    /*挂起 HUB 后，进入 D3 etc 前被调用*/
    int (*pci_suspend) (struct usb_hcd *hcd, pm_message_t message);
    /*在进入 D0 (etc)后, 恢复 Hub 前使用*/
    int (*pci_resume) (struct usb_hcd *hcd);
    /*使 HCD 停止写内存和进行 I/O 操作*/
    void(*stop) (struct usb_hcd *hcd);
    /*关闭 HCD*/
    void(*shutdown) (struct usb_hcd *hcd);
    /*返回当前的帧数*/
    int (*get_frame_number) (struct usb_hcd *hcd);
    /*管理 I/O 请求和设备状态*/
    int (*urb_enqueue)(struct  usb_hcd  *hcd,  struct  urb  *urb,  gfp_t
    mem flags);
    int (*urb_dequeue)(struct usb_hcd *hcd, struct urb *urb, int status);
    /*释放端点资源*/
    void    (*endpoint_disable)(struct usb_hcd *hcd, struct usb_host_endp-
    oint *ep);
    /*支持根 HUB*/
    int (*hub_status_data) (struct usb_hcd *hcd, char *buf);
    int (*hub_control) (struct usb_hcd *hcd, u16 typeReq, u16 wValue, u16
    wIndex,
                char *buf, u16 wLength);
    int (*bus_suspend)(struct usb_hcd *);
    int (*bus_resume)(struct usb_hcd *);
    int (*start_port_reset)(struct usb_hcd *, unsigned port_num);
        /*强制将高速端口转化为全速 companion*/
    void(*relinquish_port)(struct usb_hcd *, int);
        /*是否有端口向 companion 转化*/
    int (*port_handed_over)(struct usb_hcd *, int);
};
```

在 Linux 内核中，使用函数 usb_create_hcd()创建 HCD：

```
struct usb_hcd *usb_create_hcd (const struct hc_driver *driver, struct
device *dev, char *bus_name);
```

在 Linux 内核中，使用函数 usb_add_hcd()和 usb_remove_hcd()增加和移除 HCD：

```
int usb_add_hcd(struct usb_hcd *hcd, unsigned int irqnum, unsigned long
irqflags);
void usb_remove_hcd(struct usb_hcd *hcd);
```

### 8.2.3　OHCI 主机控制器驱动

OHCI HCD 驱动属于主机驱动 HCD 的实例，该结构体中指定了与主机控制器通信的 I/O 内存、主存、主机控制器的队列信息和队列数据管理、驱动状态信息和 ID 等。

```
struct ohci_hcd {
    spinlock_t      lock;
    /*与主机控制器通信的 I/O 内存（DMA 一致）*/
     struct ohci_regs __iomem *regs;
    /*与主机控制器通信的主存（DMA 一致）*/
    struct ohci_hcca    *hcca;
    dma_addr_t          hcca_dma;
    struct ed       *ed_rm_list;            /*指向将被移除的 OHCI 端点*/
    struct ed       *ed_bulktail;           /*批量队列尾*/
    struct ed       *ed_controltail;        /*控制队列尾*/
    struct ed       *periodic [NUM_INTS];   /*shadow int_table*/
     /*OTG 控制器和收发器需要软件交互，其他的外部收发器应该是软件透明的*/
    struct otg_transceiver  *transceiver;
    void (*start_hnp)(struct ohci_hcd*ohci);
    /*队列数据的内存管理*/
    struct dma_pool     *td_cache;
    struct dma_pool     *ed_cache;
    struct td       *td_hash [TD_HASH_SIZE];
    struct list_head pending;

    /*驱动状态*/
    int         num_ports;
    int         load [NUM_INTS];
    u32         hc_control;                 /*主机控制器控制寄存器的复制*/
    unsigned long       next_statechange;   /*挂起/恢复*/
    u32         fminterval;                 /*被保存的寄存器*/
    unsigned    autostop:1;                 /*rh auto stopping/stopped*/
    unsigned long       flags;              /*for HC bugs*/
/*各厂家芯片 ID 定义*/
#define OHCI_QUIRK_AMD756   0x01            /*erratum #4*/
#define OHCI_QUIRK_SUPERIO  0x02            /*natsemi*/
#define OHCI_QUIRK_INITRESET    0x04        /*SiS, OPTi, ...*/
#define OHCI_QUIRK_BE_DESC  0x08            /*BE descriptors*/
#define OHCI_QUIRK_BE_MMIO  0x10            /*BE registers*/
#define OHCI_QUIRK_ZFMICRO  0x20            /*Compaq ZFMicro chipset*/
#define OHCI_QUIRK_NEC      0x40            /*lost interrupts*/
```

```
#define OHCI_QUIRK_FRAME_NO     0x80    /*no big endian frame_no shift*/
#define OHCI_QUIRK_HUB_POWER    0x100
                                    /*distrust firmware power/oc setup*/
#define OHCI_QUIRK_AMD_ISO  0x200       /*ISO transfers*/
    // there are also chip quirks/bugs in init logic
    struct work_struct nec_work;        /*Worker for NEC quirk*/
    /*Needed for ZF Micro quirk*/
    struct timer_list unlink_watchdog;
    unsigned        eds_scheduled;
    struct ed       *ed_to_check;
    unsigned        zf_delay;
#ifdef DEBUG
    struct dentry       *debug_dir;
    struct dentry       *debug_async;
    struct dentry       *debug_periodic;
    struct dentry       *debug_registers;
#endif
};
```

## 8.2.4　S3C24XX OHCI 主机控制器驱动实例

本节将以 S3C24XX OHCI 主机控制器驱动的生命周期过程介绍 OHCI 主机控制器。

（1）S3C24XX OHCI 主机驱动安装系统后，系统自动识别 S3C24XX OHCI 主机驱动。

```
static struct platform_driver ohci_hcd_s3c2410_driver = {
    .probe      = ohci_hcd_s3c2410_drv_probe,
    .remove     = ohci_hcd_s3c2410_drv_remove,
    .shutdown   = usb_hcd_platform_shutdown,
    .driver     = {
        .owner  = THIS_MODULE,
        .name   = "s3c2410-ohci",
    },
};
```

（2）识别到主机类型为 S3C24XX OHCI 后，自动调用驱动函数 ohci_hcd_s3c2410_drv_probe()。在该函数中会调用 usb_hcd_s3c2410_probe()。

```
static int ohci_hcd_s3c2410_drv_probe(struct platform_device *pdev)
{
    return usb_hcd_s3c2410_probe(&ohci_s3c2410_hc_driver, pdev);
}
```

（3）在函数 usb_hcd_s3c2410_probe()中会调用函数 usb_create_hcd()创建主机控制器驱动实例。

```
static int usb_hcd_s3c2410_probe (const struct hc_driver *driver,
                  struct platform_device *dev)
{
    struct usb_hcd *hcd = NULL;
    int retval;
    /*配置端口 1 电源*/
    s3c2410_usb_set_power(dev->dev.platform_data, 1, 1);
    /*配置端口 2 电源*/
    s3c2410_usb_set_power(dev->dev.platform_data, 2, 1);
    /*创建 USB 主机驱动控制器*/
```

```
    hcd = usb_create_hcd(driver, &dev->dev, "s3c24xx");
    if (hcd == NULL)
        return -ENOMEM;
    /*下面为对创建的主机控制器进行初始化*/
    hcd->rsrc_start = dev->resource[0].start;
    hcd->rsrc_len   = dev->resource[0].end - dev->resource[0].start + 1;
    if (!request_mem_region(hcd->rsrc_start, hcd->rsrc_len, hcd_name)) {
        dev_err(&dev->dev, "request_mem_region failed\n");
        retval = -EBUSY;
        goto err_put;
    }
    /*获取时钟信息*/
    clk = clk_get(&dev->dev, "usb-host");
    if (IS_ERR(clk)) {
        dev_err(&dev->dev, "cannot get usb-host clock\n");
        retval = -ENOENT;
        goto err_mem;
    }
    usb_clk = clk_get(&dev->dev, "usb-bus-host");
    if (IS_ERR(usb_clk)) {
        dev_err(&dev->dev, "cannot get usb-host clock\n");
        retval = -ENOENT;
        goto err_clk;
    }
    s3c2410_start_hc(dev, hcd);
    /*将主机控制器的 I/O 地址空间映射到内存的虚拟地址空间，便于后面的访问*/
    hcd->regs = ioremap(hcd->rsrc_start, hcd->rsrc_len);
    if (!hcd->regs) {
        dev_err(&dev->dev, "ioremap failed\n");
        retval = -ENOMEM;
        goto err_ioremap;
    }
    /*初始化主机控制器*/
    ohci_hcd_init(hcd_to_ohci(hcd));
    /*USB 主机控制器初始化和注册*/
    retval = usb_add_hcd(hcd, dev->resource[1].start, IRQF_DISABLED);
    return 0;
    /*省略了出错处理部分*/
}
```

（4）函数 usb_create_hcd()创建主机控制器驱动实例。

```
struct usb_hcd *usb_create_hcd (const struct hc_driver *driver, struct
device *dev, const char *bus_name)
{
    struct usb_hcd *hcd;
    /*为主机控制器分配空间*/
    hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, GFP_KERNEL);
    if (!hcd) {
        dev_dbg (dev, "hcd alloc failed\n");
        return NULL;
    }
    /*设置设备 dev 的 driver_data 字段*/
    dev_set_drvdata(dev, hcd);
    /*初始化主机控制器的计数器*/
    kref_init(&hcd->kref);
    /*初始化 usb_bus 结构体*/
    usb_bus_init(&hcd->self);
```

```
    /*设置结构体 usb_bus 信息*/
    hcd->self.controller = dev;
    hcd->self.bus_name = bus_name;
    hcd->self.uses_dma = (dev->dma_mask != NULL);
    /*初始化主机控制器根 HUB 的 polling 时间*/
    init_timer(&hcd->rh_timer);
    hcd->rh_timer.function = rh_timer_func;
    hcd->rh_timer.data = (unsigned long) hcd;
#ifdef CONFIG_PM
    INIT_WORK(&hcd->wakeup_work, hcd_resume_work);
#endif
    /*指定驱动，完成了主机控制初始化后，绑定驱动到主机控制器上*/
    hcd->driver = driver;
    hcd->product_desc = (driver->product_desc) ? driver->product_desc :
            "USB Host Controller";
    return hcd;
}
```

（5）结构体 ohci_s3c2410_hc_driver 定义了 S3C2410 主机控制器驱动。

```
static const struct hc_driver ohci_s3c2410_hc_driver = {
    .description =      hcd_name,
    .product_desc =     "S3C24XX OHCI",
    .hcd_priv_size =    sizeof(struct ohci_hcd),
    /*通用硬件连接*/
    .irq =          ohci_irq,
    .flags =        HCD_USB11 | HCD_MEMORY,

    /*基本生命周期操作*/
    .start =        ohci_s3c2410_start,
    .stop =         ohci_stop,
    .shutdown =     ohci_shutdown,
    /*管理 I/O 请求和相关的资源*/
    .urb_enqueue =      ohci_urb_enqueue,
    .urb_dequeue =      ohci_urb_dequeue,
    .endpoint_disable = ohci_endpoint_disable,
    /*调度支持*/
    .get_frame_number = ohci_get_frame,
    /*根 HUB 支持*/
    .hub_status_data =  ohci_s3c2410_hub_status_data,
    .hub_control =      ohci_s3c2410_hub_control,
#ifdef  CONFIG_PM
    .bus_suspend =      ohci_bus_suspend,
    .bus_resume =       ohci_bus_resume,
#endif
    .start_port_reset = ohci_start_port_reset,
};
```

（6）当函数 s3c2410_start_hc()被调用时，平台信息被传递。执行 S3C2410 主机驱动的 start 时会自动执行函数 ohci_s3c2410_start()。

```
static int ohci_s3c2410_start (struct usb_hcd *hcd)
{
    struct ohci_hcd *ohci = hcd_to_ohci (hcd);
    int ret;
    if ((ret = ohci_init(ohci)) < 0)        //初始化 ohci 主机控制器
        return ret;
    if ((ret = ohci_run (ohci)) < 0) {      //运行 ohci 主机控制器
        err ("can't start %s", hcd->self.bus_name);
```

```
        ohci_stop (hcd);
        return ret;
    }
    return 0;
}
```

完成前面 6 步就将 ohci 主机运行起来了。其注销过程与主机驱动注册过程类似。注销时执行驱动移除函数 ohci_hcd_s3c2410_drv_remove()，调用函数 usb_hcd_s3c2410_remove()，在该函数中移除主机控制器实例，执行 s3c2410_stop_hc()，并释放资源。

```
static void usb_hcd_s3c2410_remove (struct usb_hcd *hcd, struct
platform_device *dev)
{
    usb_remove_hcd(hcd);                        //移除主机控制器
    s3c2410_stop_hc(dev);                       //释放资源
    iounmap(hcd->regs);                         //释放 I/O 映射的内存空间
    release_mem_region(hcd->rsrc_start, hcd->rsrc_len);
                                                //释放 HCD 占用的资源
    usb_put_hcd(hcd);                           //注销 HCD 对象
}
```

# 8.3　USB 设备驱动

USB 驱动大致分为：音频设备类、通信设备类、HID（人机接口）设备类、显示设备类、海量设备类、电源设备类、打印设备类和集线器设备类。USB 骨架提供了一个最基础的 USB 驱动程序，本节将通过 USB 骨架来分析 USB 驱动的写法。

## 8.3.1　USB 骨架程序分析

在 Linux kernel 源码目录中 driver/usb/usb-skeleton.c 中描述 USB 驱动编写的主要框架。下面介绍 USB 驱动主要实现的部分。

### 1．设备驱动结构体

结构体 usb_driver 定义了驱动的名字和驱动的接口函数，在驱动注册的时候，这些信息被注册到系统中，调用系统的这些函数访问设备时，就会调用对应到驱动的相应的接口函数。

```
static struct usb_driver skel_driver = {
    .name =     "skeleton",             //驱动名字
    .probe =    skel_probe,             //驱动探针函数
    .disconnect =   skel_disconnect,    //驱动断开函数
    .suspend =      skel_suspend,       //驱动挂起函数
    .resume =   skel_resume,            //驱动恢复
    .pre_reset =    skel_pre_reset,
    .post_reset =   skel_post_reset,
    .id_table = skel_table,             //驱动支持的产品 ID 和厂家 ID
    .supports_autosuspend = 1,
};
```

### 2．文件操作结构体与设备初始化

在结构体 skel_ops 中定义了 usb-skel 设备的各种操作函数。当在 usb-skel 设备上发生相应的操作时，USB 文件系统会调用对应的函数进行处理。

```
static const struct file_operations skel_fops = {
    .owner =    THIS_MODULE,
    .read =         skel_read,      //读操作
    .write =    skel_write,         //写操作
    .open =         skel_open,      //打开操作
    .release =  skel_release,       //释放操作
    .flush =    skel_flush,         //清除操作
};
```

从 skel_driver 结构可以知道 usb-skel 设备的初始化函数是 skel_probe()函数。设备初始化过程主要包括探测设备类型、分配 USB 设备使用的 URB 资源、注册 USB 设备操作函数等。skel_class 结构体记录了 usb-skel 设备信息。

```
static struct usb_class_driver skel_class = {
    .name =     "skel%d",
    .fops =     &skel_fops,
    .minor_base =   USB_SKEL_MINOR_BASE,
};
```

name 变量中采用%d 通配符表示十进制整数。当一个新的 usb-skel 类型的设备被接入 USB 总线后，会按照子设备编号自动为该设备设置设备名称。fops 是设备的文件操作结构体变量。

### 3．USB骨架程序模块的注册和注销

当使用 insmod skeleton.ko 加载模块时，函数 usb_skel_init()被调用，在该函数中调用函数 usb_register()注册设备驱动。

```
static int __init usb_skel_init(void)
{
    int result;
    /*注册 usb 驱动*/
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);
    return result;
}
```

当使用 rmmod skeleton 卸载模块时，usb_skel_exit()函数被调用，在该函数中调用 usb_deregister()函数注销模块驱动。

```
static void __exit usb_skel_exit(void)
{
    /*注销 USB 驱动*/
    usb_deregister(&skel_driver);
}
当使用 rmmod skeleton 卸载模块时，usb_skel_exit()函数被调用，调用 usb_deregister()
函数注销模块驱动。
```

#### 4．USB骨架驱动所支持的设备

在数组 skel_table []中指定驱动所支持设备的 VENDOR_ID 和 PRODUCT_ID，一些类似设备的驱动工作可以通过在该数组后添加该设备的 NEW_VENDOR_ID 和 NEW_PRODUCT_ID 方式实现。添加方式为：

```
/*定义驱动程序所支持的厂家 ID 和产品 ID*/
#define USB_SKEL_VENDOR_ID 0xfff0
#define USB_SKEL_PRODUCT_ID0xfff0
/*表中包含驱动所支持的所有厂家 ID 和产品 ID*/
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { USB_DEVICE(NEW_VENDOR_ID, NEW_PRODUCT_ID) },
                                        //此处添加支持的新设备的 ID 号
    { }                 /*Terminating entry*/
};
```

#### 5．USB骨架驱动探测函数

当一个设备被安装或者有设备插入后，USB 核心认为该驱动程序应该进行处理时，探测函数被调用，探测函数检查传递给它的设备信息，确定驱动程序是否支持该设备。

```
static int skel_probe(struct usb_interface *interface, const struct
usb_device_id *id)
{
    struct usb_skel *dev;
    struct usb_host_interface *iface_desc;
    struct usb_endpoint_descriptor *endpoint;
    size_t buffer_size;
    int i;
    int retval = -ENOMEM;
    /*为设备分配空间并初始化*/
    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
    if (!dev) {
        err("Out of memory");
        goto error;
    }
    kref_init(&dev->kref);
    sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
    mutex_init(&dev->io_mutex);
    spin_lock_init(&dev->err_lock);
    init_usb_anchor(&dev->submitted);
    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;
    /*设置端点信息*/
    /*仅用于第一个块输入和输出端点*/
    iface_desc = interface->cur_altsetting;
    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc;

        if (!dev->bulk_in_endpointAddr &&
            usb_endpoint_is_bulk_in(endpoint)) {
            /*we found a bulk in endpoint*/
            buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
            dev->bulk_in_size = buffer_size;
```

```
            dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
            dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
            if (!dev->bulk in buffer) {
                err("Could not allocate bulk_in_buffer");
                goto error;
            }
        }
        if (!dev->bulk_out_endpointAddr &&
            usb endpoint is bulk out(endpoint)) {
            /*we found a bulk out endpoint*/
            dev->bulk out endpointAddr = endpoint->bEndpointAddress;
        }
    }
    if (!(dev->bulk in endpointAddr && dev->bulk out endpointAddr)) {
        err("Could not find both bulk-in and bulk-out endpoints");
        goto error;
    }
    /*在接口设备中保存数据指针*/
    usb set intfdata(interface, dev);
    /*注册 USB 设备*/
    retval = usb register dev(interface, &skel class);
    if (retval) {
        /*something prevented us from registering this driver*/
        err("Not able to get a minor for this device.");
        usb_set_intfdata(interface, NULL);
        goto error;
    }
    /*通知用户设备所依附的结点*/
    info("USB Skeleton device now attached to USBSkel-%d", interface->
    minor);
    return 0;
error:
    if (dev)
        /*this frees allocated memory*/
        kref_put(&dev->kref, skel_delete);
    return retval;
}
```

探针函数在设备插入后被调用，如果没有正确注册设备则提示设备不是活动的设备；如果正确安装驱动则告知用户设备所依附的结点。

### 6．USB骨架驱动断开函数

当驱动程序因为某种原因（比如被拔出）不应该控制设备时，断开函数 skel_disconnect()被调用，该函数做一些清理工作。

```
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;

    dev = usb_get_intfdata(interface);      //获得 USB 设备接口描述
    usb_set_intfdata(interface, NULL);      //设置 USB 设备接口描述无效

    /*注销 USB 设备，释放设备号*/
    usb deregister dev(interface, &skel class);

    /*防止其他 I/O 访问该设备*/
```

```
    mutex_lock(&dev->io_mutex);
    dev->interface = NULL;
    mutex_unlock(&dev->io_mutex);
    usb_kill_anchored_urbs(&dev->submitted);

    /*减少引用计数*/
    kref_put(&dev->kref, skel_delete);
    info("USB Skeleton #%d now disconnected", minor);
}
```

当 USB 设备与主机断开时，调用驱动程序的断开 skel_disconnect()函数，并释放 usb-skel 设备用到的资源。函数 skel_disconnect()首先获取 USB 设备接口描述，然后再将其设置为无效。接着调用 usb_deregister_dev()函数注销 USB 设备的操作描述符，注销操作本身需要加锁。注销设备描述符后，更新内核对 usb-skel 设备的引用计数。

## 8.3.2　USB 驱动移植的时钟设置

在移植 USB 驱动的过程中，如果 USB 时钟设置不正确会发生超时错误。通过查看 S3C2440A 芯片资料的 USB 时钟控制，在 S3C2440A 内需要 PLL（upll）产生 48MHz 的时钟给 USB，UCLK 直到 PLL（UPLL）被配置才可以提供。

由 7.3.2 节提供的公式，输出时钟频率 UPLL 相对于参考输入时钟频率 Fin 的计算公式如下：

$$Upll = (m \times Fin)/(p \times 2^s)$$

$$M = MDIV + 8$$

$$p = PDIV + 2$$

$$s = SDIV$$

修改 mach-mini2440.c 文件，在该文件中添加如下代码：

```
#include <mach/regs-clock.h>
#include <mach/usb-control.h>
#include <linux/device.h>
#include <linux/delay.h>
static struct s3c2410_hcd_info usb_mini2440_info = {
    .port[0]        = {
        .flags  = S3C_HCDFLG_USED
    }
};

/*设置 USB 时钟直到设置的频率稳定*/
int usb_mini2440_init(void)
{
       unsigned long upllvalue = (0x78<<12)|(0x02<<4)|(0x03);
                                                //设置 USB 时钟频率为 48MHz
       printk("USB Control, (c) 2009 mini2440\n");
       s3c_device_usb.dev.platform_data = &usb_mini2440_info;
       while(upllvalue!=__raw_readl(S3C2410_UPLLCON))
       {
                raw_writel(upllvalue,S3C2410_UPLLCON);
              mdelay(1);
       }
       return 0;
}
```

USB 时钟的计算过程为：

$$m=0x78（120）+8=128$$

$$p=0x02（2）+2=4$$

$$s=0x03（3）=3$$

$$Upll=（128\times 12MHz）/（4\times 2^3）=48MHz$$

```
static void __init mini2440_map_io(void)
{
    s3c24xx_init_io(mini2440_iodesc, ARRAY_SIZE(mini2440_iodesc));
    s3c24xx_init_clocks(12000000);
    s3c24xx_init_uarts(mini2440_uartcfgs,
ARRAY_SIZE(mini2440_uartcfgs));
    usb_mini2440_init();          //添加 USB 时钟设置函数
}
```

通过对 USB 时钟进行修改后，将内核移植到 ARM 板上后不会出现超时错误。在后面的 USB 驱动实例中详细介绍 USB 驱动移植的步骤。

下面将通过具体驱动实例的代码分析、内核配置和移植过程，来深化了解 USB 驱动理论。8.4 节将在 ARM 平台上移植简单的 USB 鼠标驱动和 U 盘驱动。

# 8.4　USB 鼠标键盘驱动

USB 鼠标和键盘驱动程序在 Linux 源代码中已经存在，只需在编译内核时选上即可。USB 鼠标的源代码文件为 usbmouse.c，USB 键盘的源代码文件为 usbkbd.c。

## 8.4.1　USB 鼠标驱动代码分析

鼠标输入 HID 类型，其数据传输采用中断 URB，鼠标端点类型为 in。USB 鼠标驱动的主要部分为其探针函数 usb_mouse_probe()和中断函数 usb_mouse_irq()。

```
static int usb_mouse_probe(struct usb_interface *intf, const struct
usb_device_id *id)
{
    /*接口结构体包含于设备结构体中，interface_to_usbdev 是通过接口结构体获得它的设
    备结构体。*/
    struct usb_device *dev = interface_to_usbdev(intf);
    /*usb_host_interface 是用于描述接口设置的结构体，内嵌在接口结构体
    usb_interface 中。*/
    struct usb_host_interface *interface;
    /*usb_endpoint_descriptor 是端点描述符结构体，内嵌在端点结构体
    usb_host_endpoint 中，而端点结构体内嵌在接口设置结构体中。*/
    struct usb_endpoint_descriptor *endpoint;
    struct usb_mouse *mouse;
    struct input_dev *input_dev;
    int pipe, maxp;
    int error = -ENOMEM;

    interface = intf->cur_altsetting;
```

```
/*鼠标端点数只能为 1*/
if (interface->desc.bNumEndpoints != 1)
    return -ENODEV;

endpoint = &interface->endpoint[0].desc;
/*鼠标端点类型只能为 in*/
if (!usb_endpoint_is_int_in(endpoint))
    return -ENODEV;
/*返回对应端点能够传输的最大的数据包，鼠标的返回的最大数据包为 4 个字节。*/
pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
/*为 mouse 设备结构体分配内存*/
mouse = kzalloc(sizeof(struct usb_mouse), GFP_KERNEL);
input_dev = input_allocate_device();
if (!mouse || !input_dev)
    goto fail1;
/*usb_buffer_alloc 申请内存空间用于数据传输，data 为指向该空间的地址，data_dma
则是这块内存空间的 dma 映射，即这块内存空间对应的 dma 地址。在使用 dma 传输的情况下，
则使用 data_dma 指向的 dma 区域，否则使用 data 指向的普通内存区域进行传输。
GFP_ATOMIC 表示不等待，GFP_KERNEL 是普通的优先级，可以睡眠等待，由于鼠标使用中
断传输方式，不允许睡眠状态，data 又是周期性获取鼠标事件的存储区，因此使用
GFP_ATOMIC 优先级，如果不能分配到内存则立即返回 0。*/
mouse->data = usb_buffer_alloc(dev, 8, GFP_ATOMIC, &mouse->data_dma);
if (!mouse->data)
    goto fail1;
/*usb_alloc_urb 为 urb 结构体申请内存空间，第一个参数表示等时传输时需要传送包
的数量，其他传输方式则为 0。申请的内存将通过 usb_fill_int_urb()函数进行填充。*/
mouse->irq = usb_alloc_urb(0, GFP_KERNEL);
if (!mouse->irq)
    goto fail2;
/*填充 usb 设备结构体和输入设备结构体*/
mouse->usbdev = dev;
mouse->dev = input_dev;

if (dev->manufacturer)
    strlcpy(mouse->name, dev->manufacturer, sizeof(mouse->name));

if (dev->product) {
    if (dev->manufacturer)
        strlcat(mouse->name, " ", sizeof(mouse->name));
    strlcat(mouse->name, dev->product, sizeof(mouse->name));
}
/*输出鼠标的名称、厂商 ID 和产品 ID*/
if (!strlen(mouse->name))
    snprintf(mouse->name, sizeof(mouse->name),
         "USB HIDBP Mouse %04x:%04x",
         le16_to_cpu(dev->descriptor.idVendor),
         le16_to_cpu(dev->descriptor.idProduct));
/*usb_make_path 用来获取 USB 设备在 Sysfs 中的路径，格式为 usb-usb 总线号-路
径名。*/
usb_make_path(dev, mouse->phys, sizeof(mouse->phys));
strlcat(mouse->phys, "/input0", sizeof(mouse->phys));
/*将鼠标设备的名称赋给鼠标设备内嵌的输入子系统结构体*/
input_dev->name = mouse->name;
/*将鼠标设备的设备结点名赋给鼠标设备内嵌的输入子系统结构体*/
input_dev->phys = mouse->phys;
```

```
    usb_to_input_id(dev, &input_dev->id);
    input_dev->dev.parent = &intf->dev;
    /*evbit 用来描述事件，EV_KEY 是按键事件，EV_REL 是相对坐标事件*/
    input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REL);
    /*keybit 表示键值，包括左键、右键和中键*/
    input_dev->keybit[BIT_WORD(BTN_MOUSE)] = BIT_MASK(BTN_LEFT) |
        BIT_MASK(BTN_RIGHT) | BIT_MASK(BTN_MIDDLE);
    /*relbit 表示相对坐标值*/
    input_dev->relbit[0] = BIT_MASK(REL_X) | BIT_MASK(REL_Y);
    /*除了左键、右键和中键之外的其他按键*/
    input_dev->keybit[BIT_WORD(BTN_MOUSE)] |= BIT_MASK(BTN_SIDE) |
        BIT_MASK(BTN_EXTRA);
    /*中键滚轮的滚动值*/
    input_dev->relbit[0] |= BIT_MASK(REL_WHEEL);
    /*将鼠标结构体对象赋给 input_dev*/
    input_set_drvdata(input_dev, mouse);
    /*填充输入设备 open()函数指针和 close()函数指针*/
    input_dev->open = usb_mouse_open;
    input_dev->close = usb_mouse_close;
    /*usb_fill_int_urb 用于填充 URB，将上面填充好的 mouse 结构体的数据填充进 URB
    结构体中，在 open 中递交 URB。当 URB 包含一个即将传输的 DMA 缓冲区时应该设置
    URB_NO_TRANSFER_DMA_MAP。USB 核心使用 transfer_dma 变量所指向的缓冲区，而不
    是 transfer_buffer 变量所指向的缓冲区。URB_NO_SETUP_DMA_MAP 用于 Setup 包，
    URB_NO_TRANSFER_DMA_MAP 用于所有 Data 包。*/
    usb_fill_int_urb(mouse->irq, dev, pipe, mouse->data,
             (maxp > 8 ? 8 : maxp),
             usb_mouse_irq, mouse, endpoint->bInterval);
    mouse->irq->transfer_dma = mouse->data_dma;
    mouse->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
    /*向系统注册输入设备*/
    error = input_register_device(mouse->dev);
    if (error)
        goto fail3;
   /*一般在 probe 函数中，都需要将设备相关信息保存在一个 usb_interface 结构体中，以
    便以后通过 usb_get_intfdata 获取使用。这里鼠标设备结构体信息将保存在 intf 接口结
    构体内嵌设备结构体中的 driver_data 数据成员中，即 intf->dev->dirver_data =
    mouse。*/
    usb_set_intfdata(intf, mouse);
    return 0;
   /*发生错误时释放资源*/
fail3:
    usb_free_urb(mouse->irq);
fail2:
    usb_buffer_free(dev, 8, mouse->data, mouse->data_dma);
fail1:
    input_free_device(input_dev);
    kfree(mouse);
    return error;
}
```

鼠标中断函数 usb_mouse_irq()代码分析如下：

```
static void usb_mouse_irq(struct urb *urb)
{
    /*urb 中的 context 指针用于为 USB 驱动程序保存数据*/
    struct usb_mouse *mouse = urb->context;
    /*mouse->data 指向的内存区域将保存着鼠标的按键和移动坐标信息*/
```

```
    signed char *data = mouse->data;
    struct input_dev *dev = mouse->dev;
    int status;

    switch (urb->status) {
    /*status 值为 0 表示 urb 成功返回，直接跳出循环把鼠标事件报告给输入子系统。*/
    case 0:          /*success*/
        break;
   /*ECONNRESET 为出错信息，表示 urb 被 usb_unlink_urb()函数 unlink 了，ENOENT
    为出错信息,表示 urb 被 usb kill urb()函数销毁了。usb kill urb 表示彻底结束 urb
    的生命周期，而 usb_unlink_urb 则是停止 urb，这个函数不等待 urb 完全终止就会返回给
    回调函数。这在运行中断处理程序时或者等待某自旋锁时非常有用，在这两种情况下是不能睡
    眠的，而等待一个 urb 完全停止很可能会出现睡眠的情况。*/
    case -ECONNRESET:   /*unlink*/
    case -ENOENT:
    case -ESHUTDOWN:
        return;
    /*-EPIPE:  should clear the halt*/
    default:         /*error*/
        goto resubmit;
    }
    /*向输入子系统汇报鼠标事件情况，以便作出反应。
    data 数组的第 0 个字节：bit 0、1、2、3、4 分别代表左、右、中、SIDE、EXTRA 键的按
    下情况；
    data 数组的第 1 个字节：表示鼠标的水平位移；
    data 数组的第 2 个字节：表示鼠标的垂直位移；
    data 数组的第 3 个字节：REL_WHEEL 位移。*/
    input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
    input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
    input report key(dev, BTN MIDDLE, data[0] & 0x04);
    input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
    input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);
    input report rel(dev, REL X,     data[1]);
    input_report_rel(dev, REL_Y,     data[2]);
    input_report_rel(dev, REL_WHEEL, data[3]);
    /*输入子系统通过这个同步信号在多个完整事件报告中区分每一次完整事件报告。报告的内
    容包括完整的鼠标事件，包括按键信息、绝对坐标信息和滚轮信息，即上面的信息。*/
    input_sync(dev);
/*系统周期性地获取鼠标的事件信息，因此在 URB 回调函数的末尾再次提交 URB 请求块，这样又
会调用新的回调函数，周而复始。在回调函数中提交 URB 只能是 GFP_ATOMIC 优先级的，因为 URB
回调函数运行于中断上下文中禁止导致睡眠的行为。而在提交 URB 过程中可能会需要申请内存、保
持信号量，这些操作或许会导致 USB core 睡眠。*/
resubmit:
    status = usb_submit_urb (urb, GFP_ATOMIC);
    if (status)
        err ("can't resubmit intr, %s-%s/input0, status %d",
                mouse->usbdev->bus->bus_name,
                mouse->usbdev->devpath, status);
}
```

## 8.4.2　USB 键盘驱动代码分析

USB 键盘的探针函数中不仅使用 usb_fill_int_urb()从而调用函数 usb_kbd_irq()提交键盘按下键的信息，而且也使用了 usb_fill_control_urb()调用函数 usb_kbd_led()控制键盘 Led

的状态。下面列出其主要代码。

```
static int usb_kbd_probe(struct usb_interface *iface,
             const struct usb_device_id *id)
{
input_dev->event = usb_kbd_event;
usb fill int urb(kbd->irq, dev, pipe,
             kbd->new, (maxp > 8 ? 8 : maxp),
             usb_kbd_irq, kbd, endpoint->bInterval);

usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0),
                (void *) kbd->cr, kbd->leds, 1,
                usb_kbd_led, kbd);
}
/*函数 usb_kbd_irq()提交键盘按键状态信息。*/
static void usb_kbd_irq(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;
    int i;
    switch (urb->status) {
    case 0:         /*success*/
        break;
    case -ECONNRESET:   /*unlink*/
    case -ENOENT:
    case -ESHUTDOWN:
        return;
    /*-EPIPE:  should clear the halt*/
    default:        /*error*/
        goto resubmit;
    }
    /*获得键盘扫描码并报告键盘事件*/
    for (i = 0; i < 8; i++)
        input report key(kbd->dev, usb kbd keycode[i + 224], (kbd->new[0]
        >> i) & 1);

    for (i = 2; i < 8; i++) {
        if (kbd->old[i] > 3 && memscan(kbd->new + 2, kbd->old[i], 6) ==
        kbd->new + 8) {
            if (usb kbd keycode[kbd->old[i]])
                input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]],
                0);
            else
                dev_info(&urb->dev->dev,
                        "Unknown key (scancode %#x) released.\n", kbd->
                        old[i]);
        }

        if (kbd->new[i] > 3 && memscan(kbd->old + 2, kbd->new[i], 6) ==
        kbd->old + 8) {
            if (usb_kbd_keycode[kbd->new[i]])
                input report key(kbd->dev, usb kbd keycode[kbd->new[i]],
                1);
            else
                dev info(&urb->dev->dev,
                        "Unknown key (scancode %#x) released.\n", kbd->
                        new[i]);
        }
    }
    input sync(kbd->dev);
    memcpy(kbd->old, kbd->new, 8);
```

```
resubmit:
    i = usb_submit_urb (urb, GFP_ATOMIC);
    if (i)
        err_hid ("can't resubmit intr, %s-%s/input0, status %d",
                 kbd->usbdev->bus->bus_name,
                 kbd->usbdev->devpath, i);
}
```

函数 usb_kbd_led()控制键盘 LED 的状态。

```
static void usb_kbd_led(struct urb *urb)
{
    struct usb_kbd *kbd = urb->context;

    if (urb->status)
        dev_warn(&urb->dev->dev, "led urb status %d received\n",
             urb->status);
/*在 usb_kbd_event 中对将更新的 LED 状态赋值给 kbd->newleds，比较 kbd->leds 和
kbd->newleds 的值，当发生变化时用 kbd->newleds 更新 kbd->leds 的值。如果两者相同则
不更新。*/
    if (*(kbd->leds) == kbd->newleds)
        return;
    *(kbd->leds) = kbd->newleds;
    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC))
        err_hid("usb_submit_urb(leds) failed");
}
```

当事件的类型为 LED 事件时，函数 usb_kbd_event()将当前的 LED 值保存在 kbd->newleds 中，以便在 usb_kbd_led 中进行比较和处理。具体代码如下：

```
static int usb_kbd_event(struct input_dev *dev, unsigned int type,
             unsigned int code, int value)
{
    struct usb_kbd *kbd = input_get_drvdata(dev);

    if (type != EV_LED)
        return -1;
/*有 LED 事件发生时对 LED 重新赋值*/
    kbd->newleds = (!!test_bit(LED_KANA,   dev->led) << 3) | (!!test_bit
    (LED_COMPOSE, dev->led) << 3) |(!!test_bit(LED_SCROLLL, dev->led) << 2)
    |  (!!test_bit(LED_CAPSL,  dev->led)  <<  1)  |(!!test_bit(LED_NUML,
    dev->led));

    if (kbd->led->status == -EINPROGRESS)
        return 0;

    if (*(kbd->leds) == kbd->newleds)
        return 0;

    *(kbd->leds) = kbd->newleds;
    kbd->led->dev = kbd->usbdev;
    if (usb_submit_urb(kbd->led, GFP_ATOMIC))
```

```
        err_hid("usb_submit_urb(leds) failed");

    return 0;
}
```

## 8.4.3　内核中添加 USB 鼠标键盘驱动

在内核配置中主要需要添加对 USB 的支持、对 HID 接口的支持、对 OHCI HCD 驱动的支持。

（1）内核的配置中包括对 HID 接口的支持，如图 8.7 所示。

（2）在设备驱动对 USB 支持中选择配置对 OHCI HCD 的支持，如图 8.8 所示。

在将内核移植到开发板时，使用 USB 鼠标时还需要开发板套件带 LED 屏，使用超级终端无法看到效果。

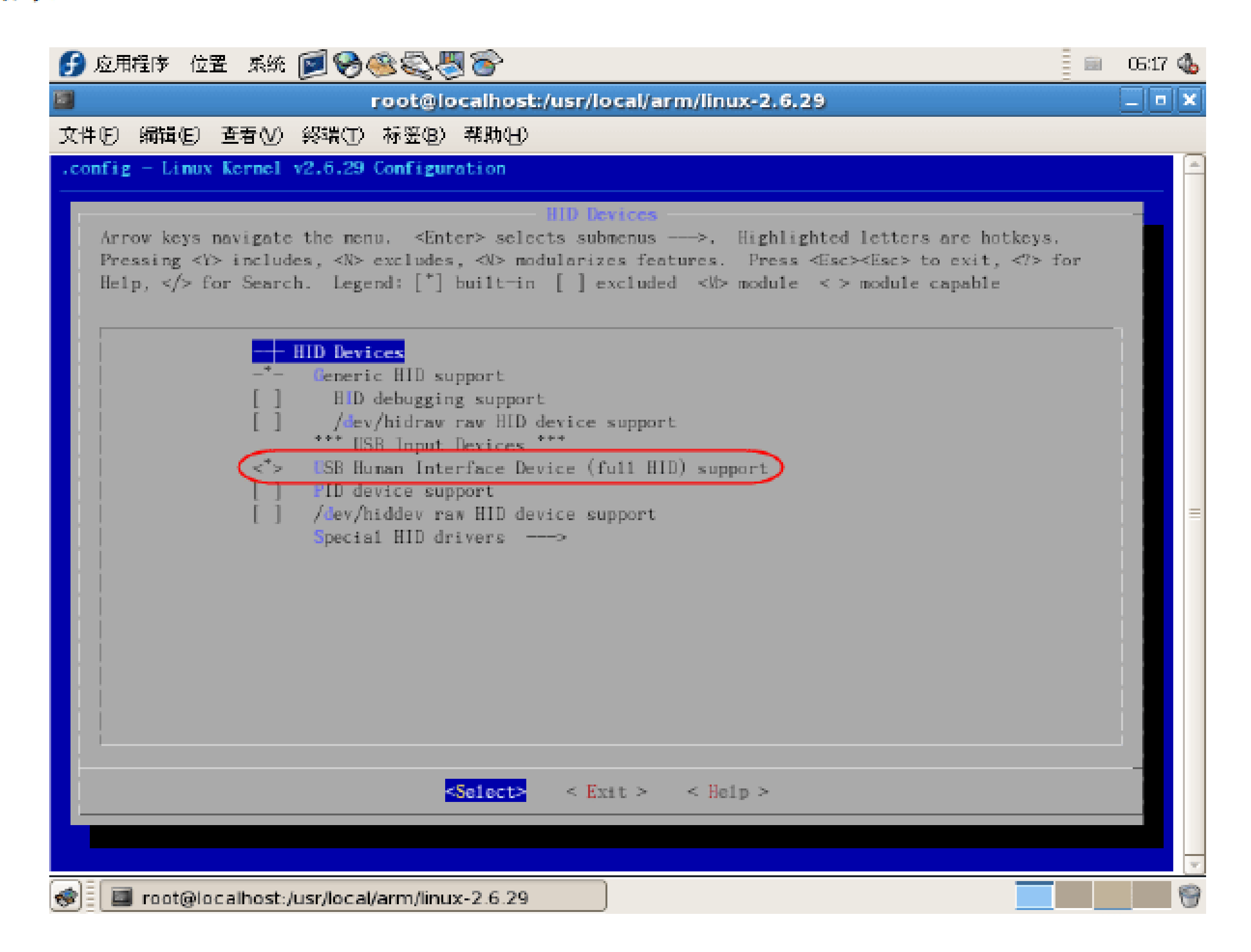



图 8.7　配置对 HID 接口的支持

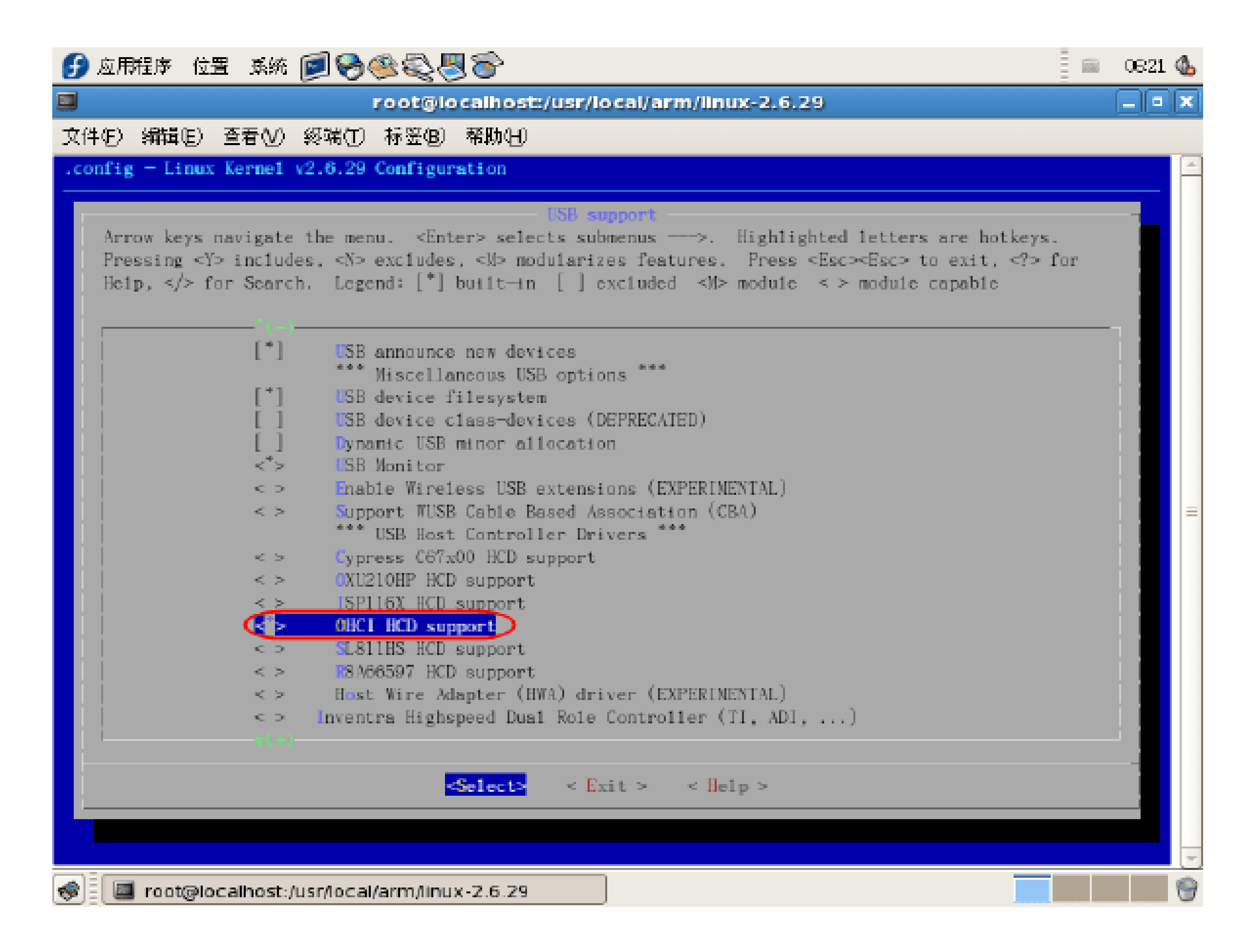



图 8.8　配置对 OHCI HCD 驱动的支持

# 8.5　U 盘驱动

USB 存储设备驱动代码在 drivers/usb/storage 目录中，而对 U 盘的访问只需在内核配置中添加支持 storage 和支持相关的文件系统对 U 盘中内容的识别，添加对字符编码的支持。

## 8.5.1　内核配置

内核中需要添加对 USB 存储设备支持、对 U 盘的存储格式支持、对中文支持的字符编码。

（1）添加对 USB Mass Storage 驱动的支持，如图 8.9 所示。

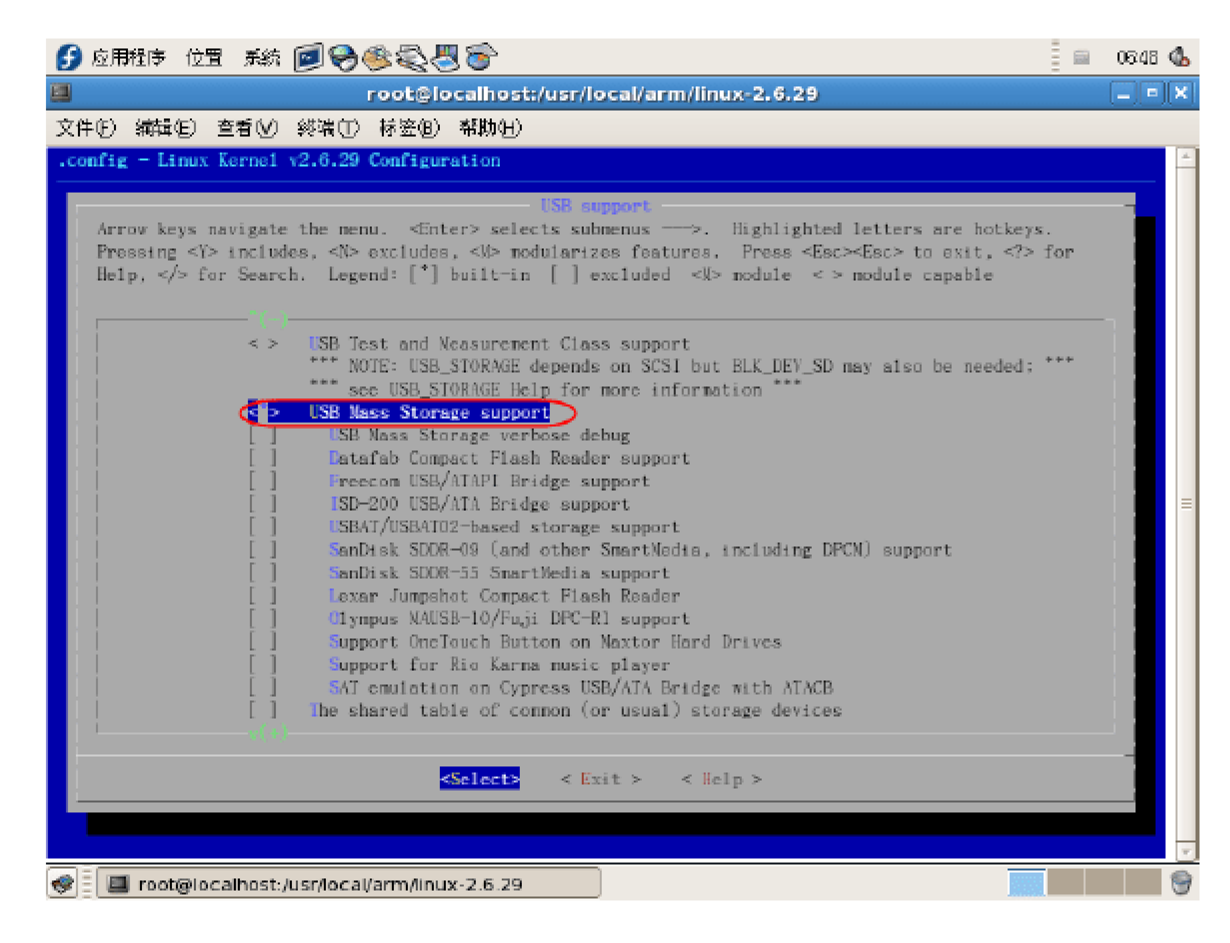



图 8.9　对 USB Mass Storage 驱动的支持

（2）配置对文件系统的支持，为了能够识别 U 盘中的文件，如图 8.10 所示。

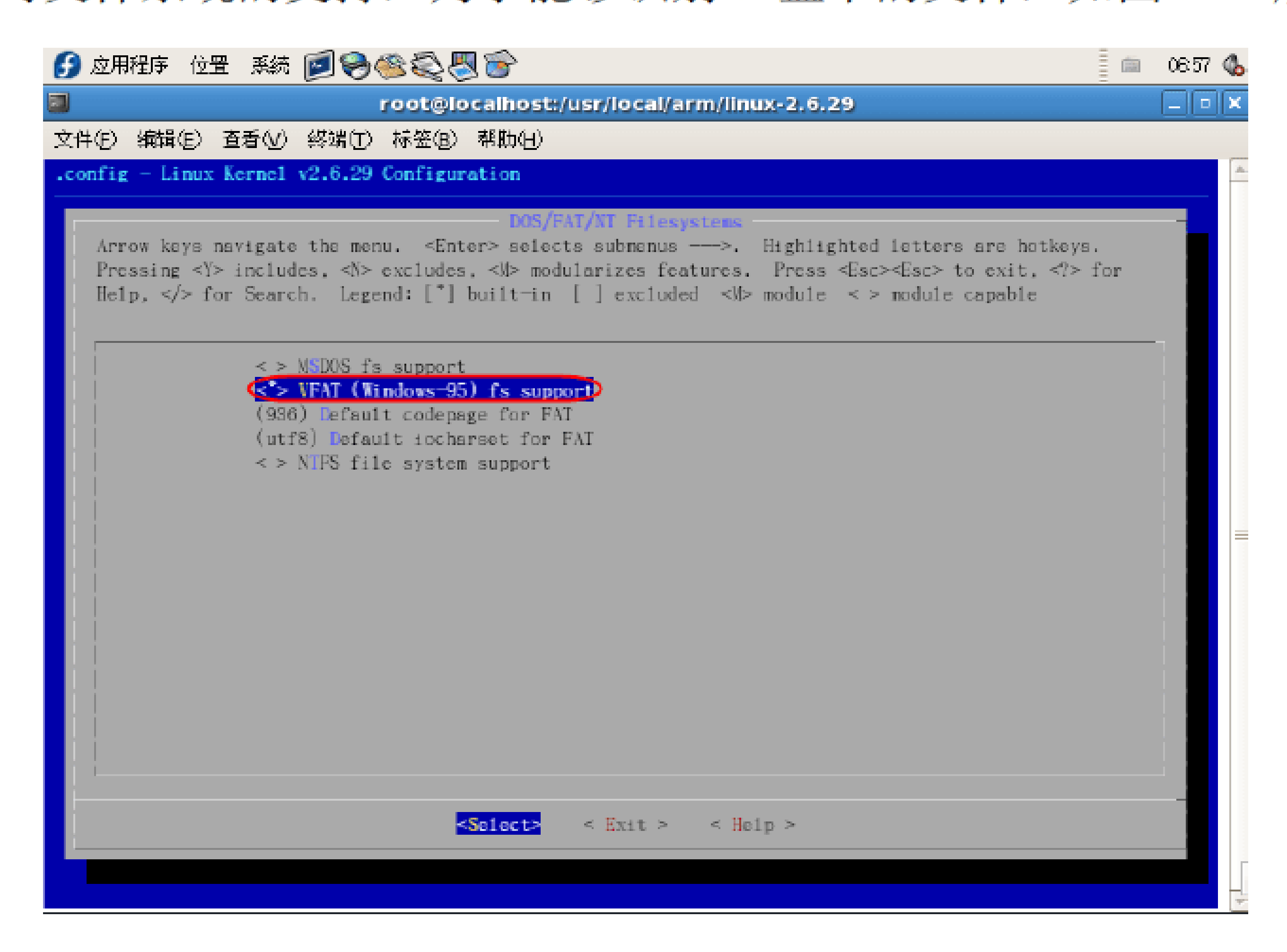



图 8.10　对文件系统的支持

（3）对字符编码和简体中文的支持，如图 8.11 和图 8.12 所示。

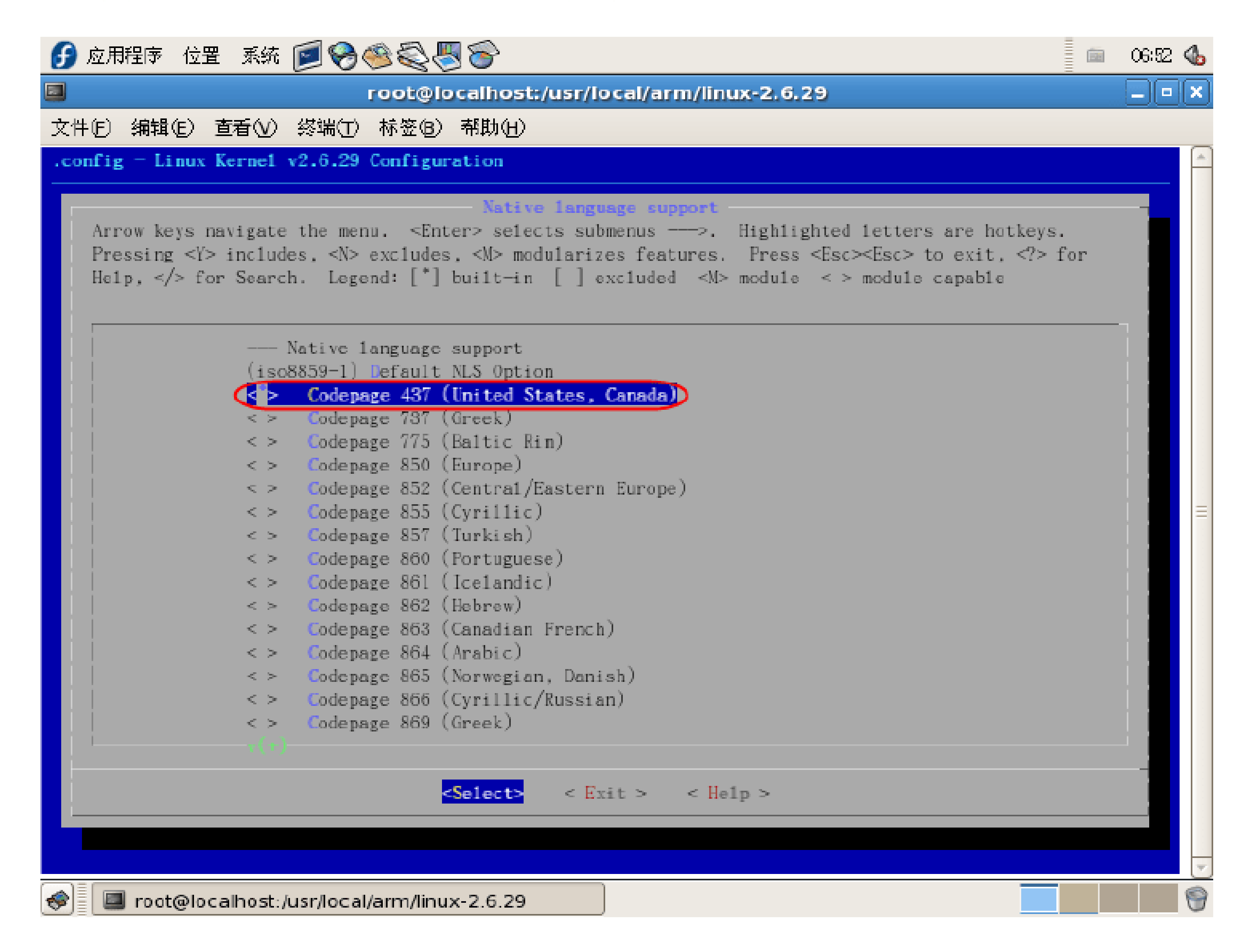



图 8.11　对编码页的支持

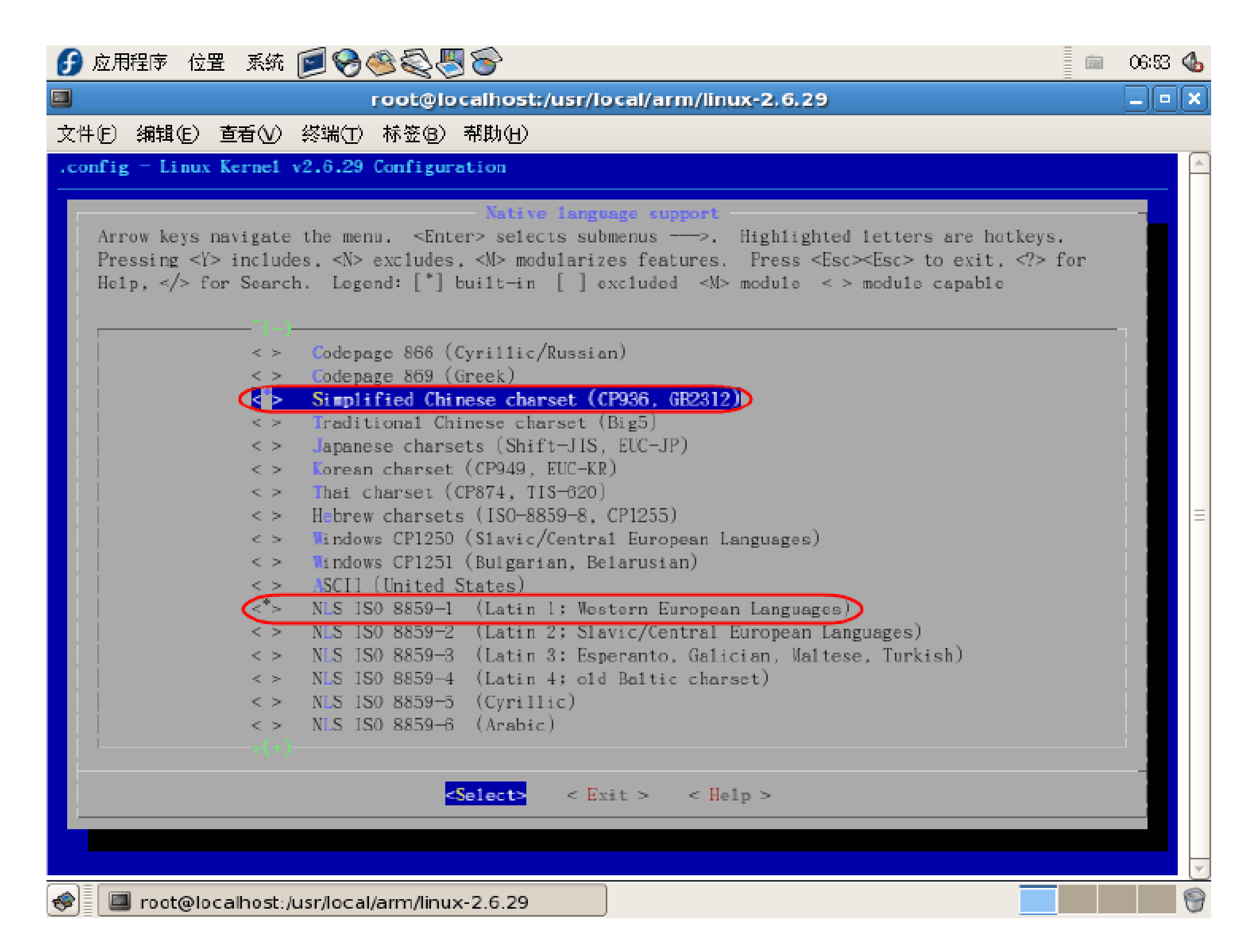



图 8.12　对简体中文和字符编码的支持

## 8.5.2　移植和测试

移植和测试时，首先需要正确识别设备并找到相应的分区，然后才能执行挂载操作。

（1）这里测试时，采用的是手机上的存储卡大小为 512M。使用 USB 线连接到 2440

开发板上。当插入到开发板上时打印如下信息：

```
usb 1-1.2: new full speed USB device using s3c2410-ohci and address 5
usb 1-1.2: New USB device found, idVendor=0e8d, idProduct=0002
usb 1-1.2: New USB device strings: Mfr=2, Product=3, SerialNumber=4
usb 1-1.2: configuration #1 chosen from 1 choice
 uba: uba1
```

这些信息表示能够正确识别 USB 设备，并读取其厂家 ID 和产品 ID，以及存储设备 uba1 信息。

执行 cat /proc/partitions 查看磁盘分区信息如下：

```
[root@FriendlyARM /]# cat  /proc/partitions
major minor  #blocks      name
   31     0       192      mtdblock0
   31     1      2048      mtdblock1
   31     2     63152      mtdblock2
  180     0    497152      uba
  180     1    497034      uba1
```

（2）挂载 U 盘。在 mnt 目录下建立 usb 目录。挂载前后的信息对比如下所示。

```
[root@FriendlyARM /]# mkdir /mnt/usb
```

挂载前分区信息：

```
[root@FriendlyARM /]# df -h
Filesystem                Size      Used  Available  Use%  Mounted on
/dev/root                61.7M     43.8M      17.9M   71%  /
tmpfs                    29.9M         0      29.9M    0%  /dev/shm
```

挂载命令：

```
[root@FriendlyARM /]# mount /dev/uba1 /mnt/usb/
```

挂载后的分区信息：

```
[root@FriendlyARM /]# df -h
Filesystem               Size      Used Available Use%   Mounted on
/dev/root               61.7M     43.8M     17.9M  71%   /
tmpfs                   29.9M         0     29.9M   0%   /dev/shm
/dev/uba1              485.3M    214.1M    271.1M  44%   /mnt/usb
```

查看 U 盘信息：

```
[root@FriendlyARM /]# ls /mnt/usb -l
-rwxr-xr-x    1 root     root         4096 Aug 13  2007 @samsung.ess
drwxr-xr-x    2 root     root        16384 Jan 25  2009 Audio
drwxr-xr-x    2 root     root        16384 Apr 29  2008 Ebook
drwxr-xr-x    3 root     root        16384 Aug 13  2007 Images
drwxr-xr-x    2 root     root        16384 Aug 13  2007 Music
drwxr-xr-x    2 root     root        16384 Jan  1  1980 My Music
drwxr-xr-x    2 root     root        16384 Aug 13  2007 Other files
drwxr-xr-x    2 root     root        16384 Dec 31  2008 Photos
drwxr-xr-x    2 root     root        16384 Aug 13  2007 Sounds
drwxr-xr-x    2 root     root        16384 Aug 13  2007 Videos
-rwxr-xr-x    1 root     root        13050 Feb  7  2010 audio_play_list.txt
drwxr-xr-x    2 root     root        16384 Dec  8  2009 software
```

# 8.6　小　　结

本章对 USB 驱动的介绍只起到入门的作用，介绍简单的 USB 鼠标键盘驱动的移植，在嵌入式平台中访问 U 盘的情况。USB 设备比较复杂，不同类型的 USB 设备包含的接口、端点及 URB 的传输类型都不相同。但不变的是 URB 的生命周期，基本包含创建、初始化、提交、USB core 与 USB 主机传递等过程。URB 的生命周期过程是设计 USB 驱动的关键。

# 第 9 章　网卡驱动程序移植

网卡是工作在物理层的网络组件，是局域网中连接计算机和传输介质的接口设备。它不仅能实现与局域网传输介质之间的物理连接和电信号匹配，还涉及帧的发送与接收、帧的封装与拆封、介质访问控制、数据的编码与解码及数据缓存的功能等。在嵌入式系统中，网卡是一种常见的外围设备，本章首先讲述以太网的基础知识，之后主要讲解针对 DM9000 网卡的驱动程序移植。

## 9.1　以太网概述

多数人将局域网（Local Area Network，LAN）和以太网（Ethernet）混为一谈，其实这是一种错误的认识。以太网是局域网技术中的一种，它和其他局域网技术比较起来，使用得更普遍、发展得更迅速，以至于人们将“以太网”当作了“局域网”的代名词。

以太网（Ethernet）是一种基带局域网规范，它是由 Xerox 公司创建并由 Xerox、Intel 和 DEC 公司联合开发的。它是当今现有局域网采用的最通用的通信协议标准。以太网络使用载波监听多路访问及冲突检测技术（CSMA/CD），并以 10M/S 的速率运行在多种类型的电缆上。以太网与 IEEE802·3 系列标准相类似，也是一种技术规范。

### 9.1.1　以太网连接

以太网技术规范中规定了以太网的拓扑结构、传输介质和工作模式，以下分别对其进行描述。

#### 1．以太网的拓扑结构

以太网拓扑结构有总线型和星型。

- 总线型：总线型网络所采用的传输介质一般也是同轴电缆（包括粗缆和细缆），不过现在也有采用光缆作为总线型传输介质的。早期以太网经常使用总线型的拓扑结构，采用同轴缆作传输介质，连接比较简单，通常在小规模的网络中不需要用到专用的网络设备。它的特点是所需的电缆较少、价格便宜、管理成本高，不易隔离故障点、采用共享的访问机制，易造成网络拥塞。因为它存在的固有缺陷，已经逐渐被以集线器和交换机为核心的星型网络所代替。
- 星型：网络中的各工作站结点设备通过一个网络集中设备（如集线器或者交换机）连接在一起，各个结点呈星状分布，这便是星型结构。其特点是管理方便、容易扩展、需要专用的网络设备作为网络的核心结点、需要更多的网线、对核心设备

的可靠性要求高。虽然星型网络需要的线缆比总线型多，但布线和连接器比总线型便宜。除此之外，星型拓扑可以通过级联的方式很方便地将网络扩展到很大的规模，因此得到了广泛的应用，被绝大多数的以太网所采用。

#### 2．以太网接口的工作模式

以太网卡可以工作在以下两种模式下：半双工与全双工。

- 半双工：半双工就是指一个时间段内只有一个动作发生，举个例子，一条窄窄的道路上，同一时刻只能有一辆车通过，当同时有两量车对开，这种情况下就只能一辆车先过，等到一辆车过后另一辆再开，这个例子很形象地说明了半双工的原理。早期的对讲机及早期集线器等设备都是基于半双工的产品。随着技术的不断进步，半双工会逐渐退出历史的舞台。
- 全双工：全双工（Full Duplex）是指网卡在发送数据的同时也能够接收数据，两者同时进行，这好像我们平时打电话一样，说话的同时也能够听到对方的声音。目前的网卡一般都支持全双工。全双工传输是采用点对点连接的，这种安排没有冲突，这是因为它们使用双绞线中两个独立的线路，这相当于没有安装新的介质就提高了带宽。标准以太网的传输效率可达到 50%～60%的带宽，双全工在两个方向上都提供了 100%的效率。

#### 3．传输介质

以太网中采用了多种传输介质，包括同轴缆、双绞线和光纤等。其中双绞线是现在最普通的传输介质，它由两条相互绝缘的铜线组成，典型直径为 1 毫米。两根线绞接在一起是为了防止电磁感应在邻近线对中产生干扰信号，它多用于从主机到集线器或交换机的连接；光纤是软而细的、利用内部全反射原理来传导光束的传输介质，它主要用于交换机间的级联和交换机到路由器间的点到点链路上。同轴缆作为早期的主要连接介质已经逐渐被淘汰了。

### 9.1.2　以太网技术概述

以下简单概述以太网的相关技术标准。

#### 1．以太网的工作原理

以太网采用带冲突检测的载波帧听多路访问（CSMA/CD）机制。以太网中其他结点都可以看到在网络中发送的所有信息，因此，称以太网是一种广播网络。

以太网的工作过程如下：

当以太网中的一台主机要传输数据的时候，它将按如下步骤进行。

（1）当一个站点想要发送数据的时候，先检测网络查看是否有其他站点正在传输，即监听信道是否空闲。

（2）如果信道忙，则等待，直到信道空闲；如果信道闲，站点就传输数据。

（3）在发送数据的同时，站点继续监听网络并确信没有其他站点在同时传输数据。因为有可能两个或多个站点都同时检测到网络空闲然后并几乎在同一时刻开始传输数据。如果两个或多个站点同时发送数据，就要产生冲突。

（4）当一个传输结点识别出一个冲突，它就发送一个拥塞信号，这个信号使得冲突的时间足够长，让其他的结点都能发现。

（5）其他结点收到拥塞信号后，都停止传输，等待一个随机产生的时间间隙（回退时间，Backoff Time）后重发。

### 2．Ethernet地址

以太网中通过给每台主机上的网络适配器（网络接口卡）分配一个唯一的通信地址标识以太网上的每台计算机，这个唯一的通信地址就是人们常说的 Ethernet 地址，通常也称为网卡的物理地址、MAC 地址。

IEEE 给网络适配器制造厂商分配了 Ethernet 地址块，这个叫厂商代号。各厂商又给自己生产的每块网络适配器分配一个唯一的 Ethernet 地址，这个叫设备编号。由于在每块网络适配器出厂时，其 Ethernet 地址都已被烧录到网络适配器中了，有时也将此地址称为烧录地址（Burned-In- Address，BIA）。

Ethernet 地址总长 48 比特，共 6 个字节。其中，前 3 个字节是 IEEE 分配给厂商的厂商代码，后 3 个字节是网络适配器编号，如图 9.1 所示。

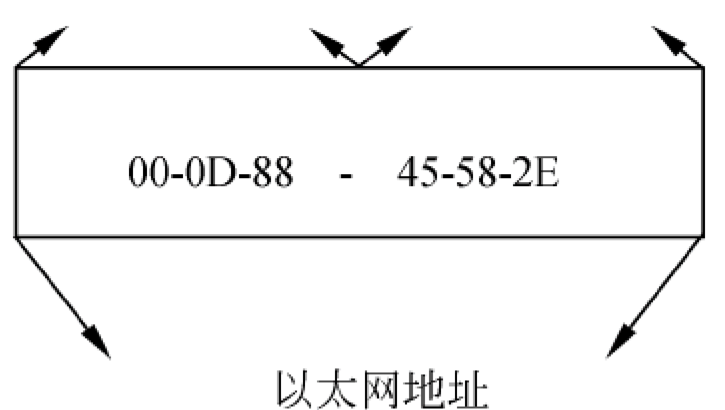



图 9.1　Ethernet 地址格式

### 3．数据链路层

数据链路层位于 OSI 参考模型中的第二层，介乎于物理层及网络层之间。数据链路层在物理层提供服务的基础上向网络层提供服务，其最基本的服务是将源计算机网络层的数据可靠地传输到相邻结点的目标机网络层。然而在局域网中，多个结点是共享传输介质的，这就必须有某种机制来决定某一个时刻，哪个设备占用传输介质来传送数据。因此，局域网的数据链路层要有介质访问控制的功能。一般数据链路层划分成两个子层，如图 9.2 所示。

- 逻辑链路控制 LLC（Logic Line Control）子层；
- 介质访问控制 MAC（Media Access Control）子层。

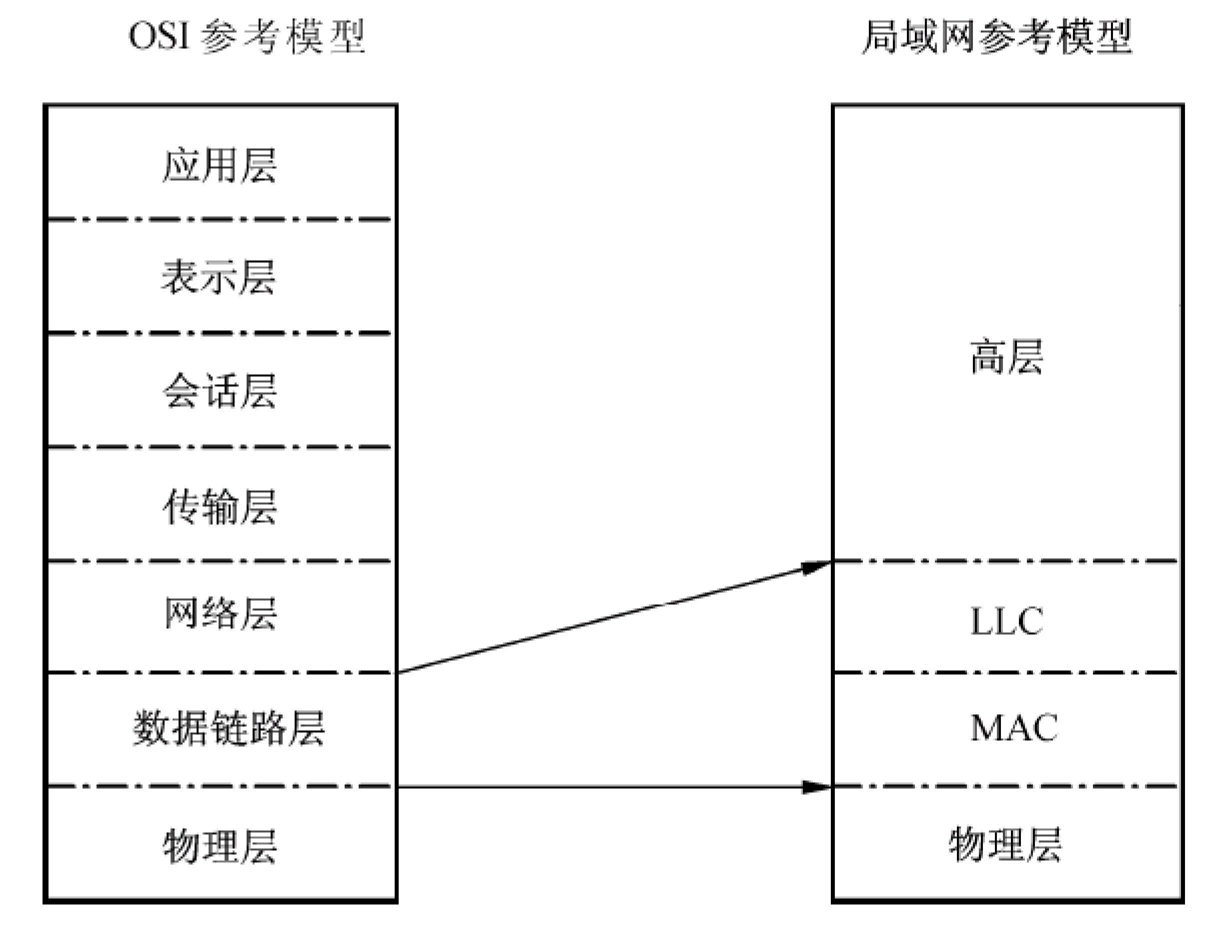



图 9.2　LLC 和 MAC 子层

其中 LLC 子层负责向其上层提供服务；LLC 是在高级数据链路控制（HDLC：High-Level Data-Link Control）的基础上发展起来的，并使用了 HDLC 规范子集。LLC 定义了 3 种数据通信操作类型。

- 类型 1：无连接。这种方式对信息的发送通常无法保证接收。
- 类型 2：面向连接。该方式提供了 4 种服务，分别是连接的建立、确认和承认响应、差错恢复（通过请求重发接收到的错误数据实现）及滑动窗口（系数：128）。通过改变滑动窗口可以提高数据传输的速率。
- 类型 3：无连接承认响应服务。

MAC 子层的主要功能包括数据帧的封装或卸装，帧的寻址与识别，帧的接收与发送，链路的管理，帧的差错控制等。MAC 子层屏蔽了不同物理链路种类的差异性；在 MAC 子层诸多功能中，特别重要的一项功能是仲裁介质的使用权，即规定站点何时可以使用共享的通信介质。局域网技术中采用具有冲突检测的载波侦听多路访问（Carrier Sense Multiple Access /Collision Detection，CSMA/CD）来控制站点访问共享介质。

## 9.1.3　以太网的帧结构

在 Ethernet 中有几种不同的帧格式，下面就简单介绍一下几种不同的帧格式及它们的差异，先分别列出各种格式的名称。

- Ethernet II 即 DIX 2.0：Xerox 与 DEC、Intel 在 1982 年制定的以太网帧格式标准，是对 Ethernet I 的补充和完善。
- Ethernet 802.3 raw：Novell 在 1983 年公布的专用以太网标准帧格式。
- Ethernet 802.3 SAP：IEEE 在 1985 年公布的 Ethernet 802.3 的 SAP 版本以太网帧格式。
- Ethernet 802.3 SNAP：IEEE 为解决 EthernetII 与 802.3 帧格式的兼容问题推出折衷的 Ethernet SNAP 格式。

在每种格式的以太网帧的开始处都有前导字符，前导字符共 64 比特（8 字节），如图 9.3 所示。这 8 字节的前导符分为两部分：前同步码和帧起始标志符。前 7 个字节为同步码，用 16 进制数 0xAA 填充；最后 1 字节是帧起始标志符，用 0xAB 填充，它标志着以太网帧的开始。前导字符用于使接收结点进行同步并做好接收数据帧的准备。

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 10101010 | 10101010 | 10101010 | 10101010 | 10101010 | 10101010 | 10101010 | 10101011 |

图 9.3　以太网帧前导字符

除前导字符之外，不同格式的以太网帧各字段定义都不相同，彼此并不兼容，以下分别介绍。

### 1．Ethernet II帧格式

Ethernet II 类型以太网帧格式如图 9.4 所示。

Ethernet II 类型以太网帧的前 12 个字节，分别表示发送该数据帧的源机 MAC 地址和接收数据帧的目标机 MAC 地址。在 MAC 地址后面的 2 个字节表示以太网帧所携带的上

层数据类型，如十六进制数 0x0800 表示 IP 协议数据，十六进制数 0x809B 表示 AppleTalk 协议数据，十六进制数 0x8138 表示 Novell 类型协议数据等。

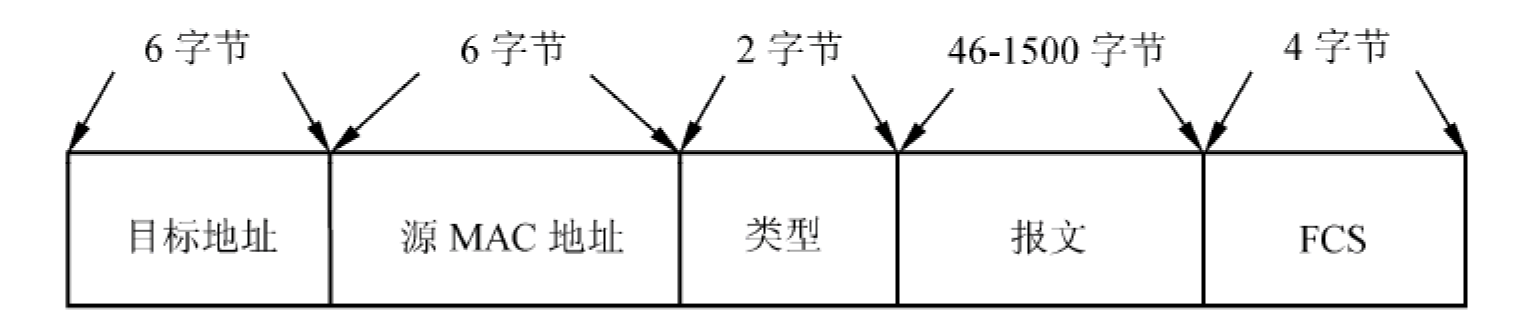



图 9.4　Ethernet II 帧格式

在类型标识后面就是以太帧所携带的真正数据了，它是不确定长度的数据块，最小长度是 46 字节，最大长度是 1500 字节。紧跟其后的是 4 字节的帧校验序列（Frame Check Sequence，FCS），一般采用 32 位 CRC 循环冗余校验，对从“目标 MAC 地址”字段到“数据”字段的数据进行校验。

### 2．Ethernet 802.3 raw帧格式

Ethernet 802.3 raw 类型以太网帧格式，如图 9.5 所示。

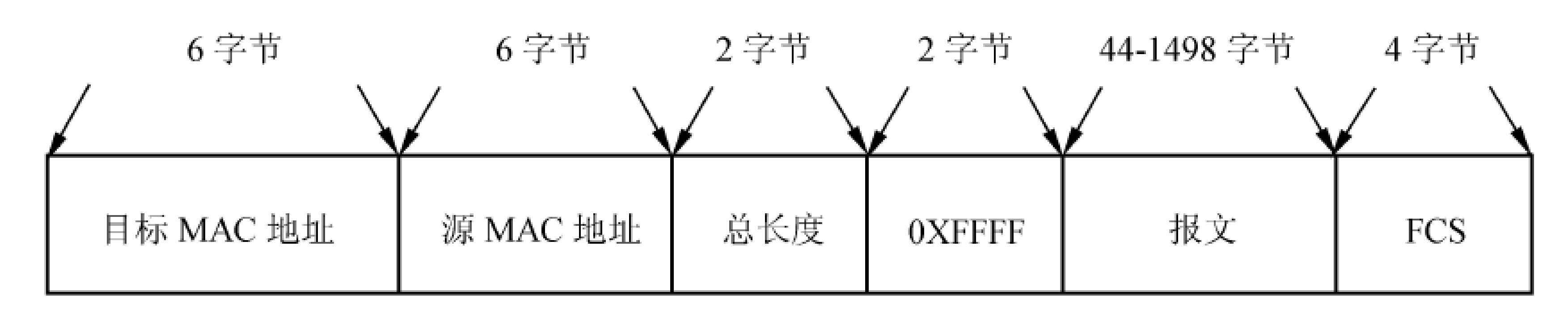



图 9.5　Ethernet 802.3 raw 帧格式

在 Ethernet 802.3 raw 类型以太网帧中，没有专用于表示所携带的上层数据类型的字段。原来 Ethernet II 类型以太网帧中用于表示所携带的上层数据类型的字被“总长度”字段所取代，其取值范围为 46～1500，它指明了其后数据域的长度。

数据字段后面的 2 个字节是固定不变的十六进制数 0xFFFF，它表示该帧为 Novell 以太类型数据帧。

### 3．Ethernet 802.3 SAP帧格式

Ethernet 802. 3 SAP 类型以太网帧格式，如图 9.6 所示。

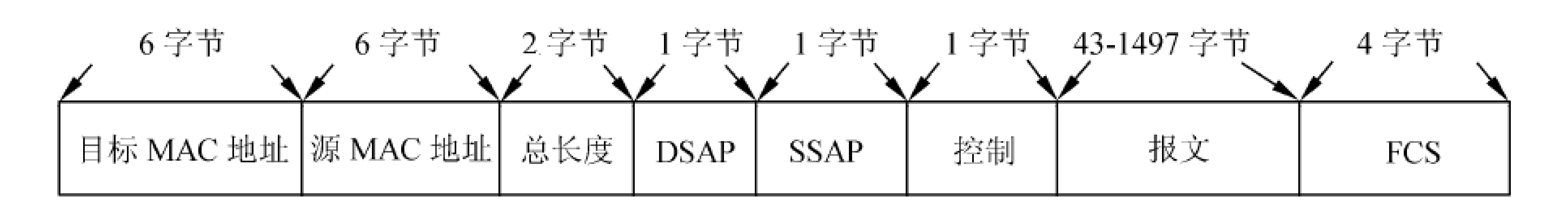



图 9.6　Ethernet 802.3 SAP 帧格式

从图中可以看出，在 Ethernet 802.3 SAP 帧中，在 Ethernet 802.3 raw 类型以太网帧格式的基础上去掉了原来 2 个字节的 0xFFFF 字段，代之以 1 个字节的 DSAP 和 1 个字节的 SSAP，同时增加了 1 个字节的“控制”字段，这三个字段构成了 802.2 逻辑链路控制（LLC）的首部。LLC 提供了无连接（LLC 类型 1）和面向连接（LLC 类型 2）的网络服务。LLC1

应用于以太网环境中，而 LLC2 应用于 IBM SNA 网络环境中。

802.2 LLC 首部包括两个服务访问点：分别是源服务访问点（SSAP）和目标服务访问点（DSAP）。它们用于表示以太网数据帧中所携带的上层数据类型，如 0x06 表示 IP 协议数据，0xE0 表示 Novell 类型协议数据，0xF0 表示 IBM NetBIOS 类型协议数据等。

1 个字节的“控制”字段基本不使用，通常被设置为 0x03，表示采用无连接服务的 802.2 无编号数据格式。

#### 4．Ethernet 802.3 SNAP帧格式

Ethernet 802.3 SNAP 类型以太网帧格式如图 9.7 所示。

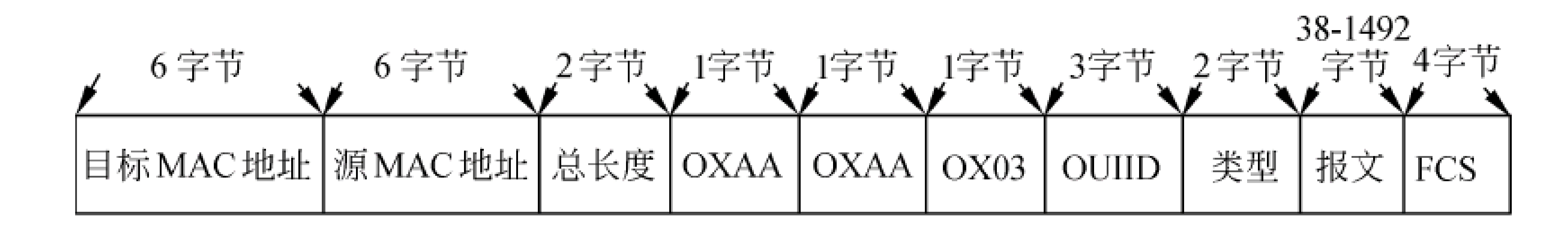



图 9.7　Ethernet 802. 3 SNAP 帧格式

Ethernet SNAP 类型数据帧格式与 802.3/802.2 类型数据帧格式的最大区别是增加了一个 5 Bytes 的 SNAP ID，其中前面 3 个字节通常与源 MAC 地址的前 3 个字节相同，为厂商代码！有时也可设为 0。后 2 个字节用来标识以太网帧所携带的上层数据类型。

## 9.2　网络设备驱动程序体系结构

网络设备是 Linux 3 种基本设备之一，它有着其他两种设备不同的特点。网络设备在系统中的作用类似于一个已挂载的块设备。块设备把自己注册到 blk_dev 数据及其他内核结构中，然后通过自己的 request()函数在发生请求时传输和接收数据块，同样网络设备也必须在特定的数据结构中注册自己，以便与外界交换数据包时被调用。

网络设备在 Linux 内核里做专门的处理。Linux 的网络系统主要是基于 BSD UNIX 的 Socket 机制。在系统和驱动程序之间定义有专门的数据结构（sk_buff）进行数据的传递。系统里支持对发送数据和接收数据的缓存，提供了流量控制机制，提供了对多协议的支持。

### 9.2.1　嵌入式 Linux 网络驱动程序介绍

Linux 网络驱动程序是 Linux 网络子系统中的一部分，位于 TCP/IP 网络体系结构的网络接口层，主要实现上层协议栈与网络设备的数据交换。Linux 的网络系统主要是基于 BSD Unix 的套接字（socket）机制，网络设备与字符设备和块设备不同，没有对应的映射到文件系统中的设备结点。

通常，Linux 驱动程序有两种加载方式：一种是静态地编译进内核，内核启动时自动加载；另一种是编写为内核模块，使用 insmod 命令将模块动态加载到正在运行的内核，不需要时可用 rmmod 命令将模块卸载。

Linux 2.6 内核引入了 kbuild 机制，将外部内核模块的编译同内核源码树的编译统一起

来，大大简化了特定参数和宏的设置。这样将编写好的驱动模块加入内核源码树，只要修改相应目录的 Kconfig 文件，把新的驱动加入内核的配置菜单中，然后需要修改相应子目录中与模块编译相关的 Kbuild Makefile，即可使新的驱动在内核源码树中被编译。在嵌入式系统驱动开发时，常常将驱动程序编写为内核模块，方便开发调试。调试完成后，就可以把驱动模块编译进内核，并重新编译出支持特定物理设备的 Linux 内核。

## 9.2.2　Linux 网络设备驱动的体系结构

如图 9.8 所示，Linux 网络驱动程序的体系结构可划分为 4 个层次，即网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层及设备物理媒介层。

Linux 内核源代码中提供了网络设备接口及以上层的代码，所以移植特定网络硬件驱动程序的主要工作就是编写设备驱动功能层的相应代码，根据底层具体硬件的特性，定义 struct net_device 这个网络设备接口类型的结构体变量，并实现其中相应操作函数及中断处理程序。

Linux 中所有的网络设备都抽象为一个统一的接口，即网络设备接口，通过 struct net_device 类型的结构体变量表示网络设备在内核中的运行情况，这里既包括回环（loopback）设备，也包含硬件网络设备接口。内核中是通过以 dev_base 为头指针的设备链表来管理所有网络设备的。

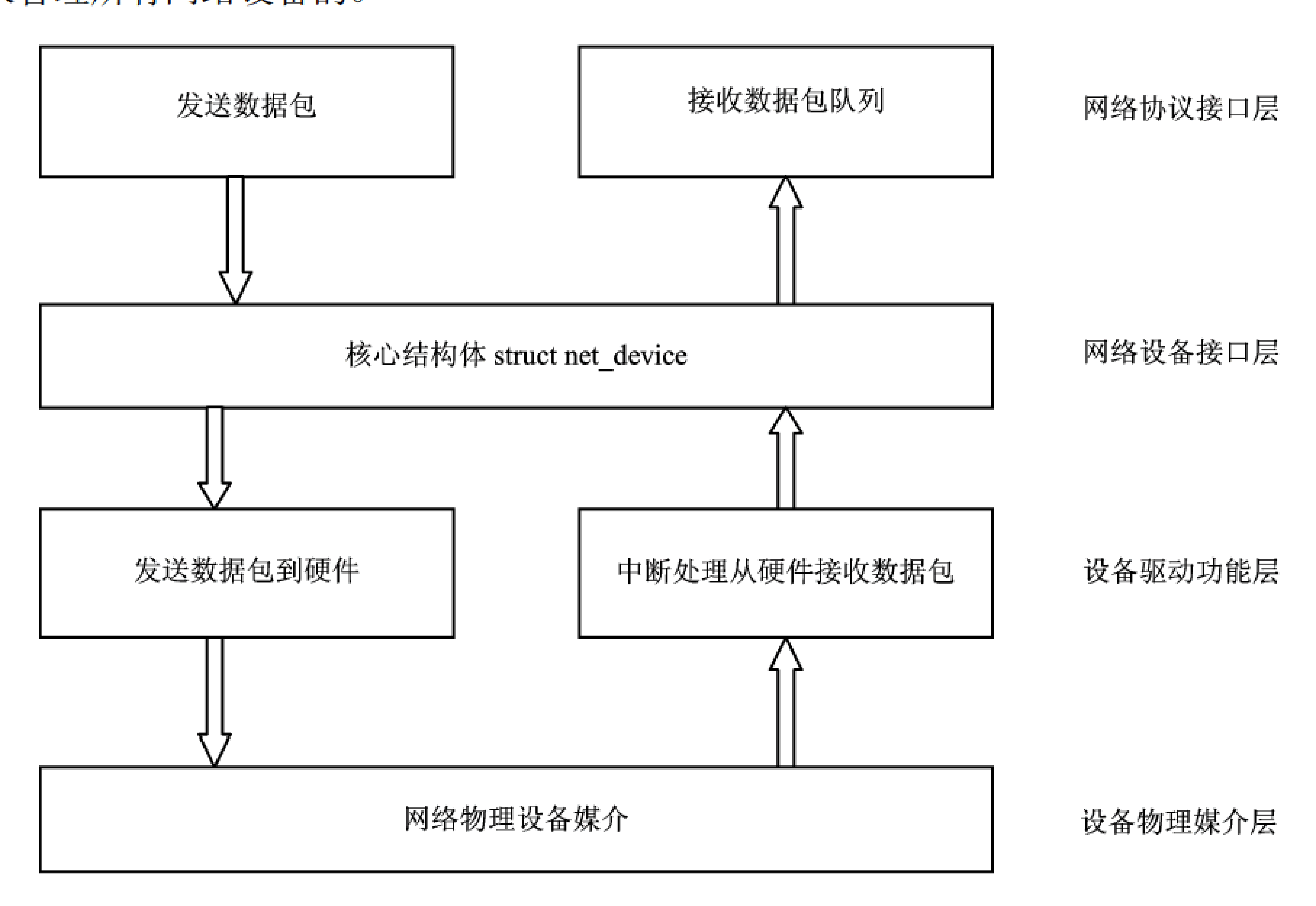



图 9.8　Linux 网络驱动程序体系结构

## 9.2.3　网络设备驱动程序编写方法

网络设备驱动程序编写包括网络设备的初始化，数据包发送和接收函数的编写及其他

相关内容，下面分别讲述。

### 1．初始化

网络设备的初始化主要是由 device 数据结构中的 init 函数指针所指的初始化函数来完成的，当内核启动或加载网络驱动模块的时候，就会调用初始化过程。在这其中将首先检测网络物理设备是否存在，这是通过检测物理设备的硬件特征来完成的，然后再对设备进行资源配置，这些工作完成之后就要构造设备的 device 数据结构，把检测到的数值对 device 中的变量初始化（这里的设备 device 结构就是前面介绍的 net_device 结构），这一步非常重要。最后调用 registre_netdevice()向 Linux 内核中注册该设备并申请内存空间。

### 2．数据包的发送与接收

数据包的发送和接收是实现 Linux 网络驱动程序中两个最重要的过程，对这两个过程处理的成功与否将直接影响到驱动程序的整体运行质量。首先，在网络设备驱动加载时，通过 device 域中的 init()函数指针调用网络设备的初始化函数对网络设备进行初始化，如果操作成功了就可以通过 device 域中的 open()函数指针调用网络设备的打开函数打开设备，再通过 device 域中的建立硬件包头函数指针 hard_header 建立硬件包头信息。

最后通过协议接口层函数 dev_queue_xmit()（详见/linux/net/core/dev.c）调用 device 域中的 hard_start_xmit()函数指针来完成数据包的发送。该函数会把保存在套接字缓冲区中的数据发送到物理设备，该缓冲区是由数据结构 sk_buff （详见/linux/include/linux/sk_buff.h）来表示的。

数据包接收是通过系统的中断机制来完成的，当有网络数据到达时，就产生中断信号，网络设备驱动功能程序就调用中断处理程序，即数据包接收函数来处理数据包的接收，然后网络协议接口层调用 netif_rx()函数（详见/linux/net/core/dev.c）把接收到的数据包传输到网络协议的上层进行处理。

### 3．实现模式

实现 Linux 网络设备驱动的功能主要有两种形式，一是通过内核进行加载，当内核启动的时候，就开始加载网络设备驱动程序，内核启动完成之后，网络驱动功能也随即实现了，再就是通过模块加载的形式。比较这两种形式，第二种形式更灵活些，在此重点对模块加载形式进行讨论。

模块设计是 Linux 中特有的技术，它使 Linux 内核功能更容易扩展。采用模块来设计 Linux 网络设备驱动程序会很轻松，并且能够形成固定的模式，任何人只要按照这个模式去设计，都能设计出优良的网络驱动程序。

先简要概述一下基于模块加载的网络驱动程序的设计步骤。首先通过模块加载命令 insmod 把网络设备驱动程序插入到内核中。然后 insmod 将调用 init_module()函数首先对网络设备的 init()函数指针初始化，再通过调用 register_netdev()函数在 Linux 系统中注册该网络设备，如果注册成功，再调用 init()函数指针所指的网络设备初始化函数对设备进行初始化，将设备的 device 数据结构插入到 dev_base 链表的尾端。最后可以通过执行模块卸载命令 rmmod 调用网络驱动程序中的 cleanup_module()函数，对网络驱动程序模块卸载。

通过模块初始化网络接口是在编译内核时做标记，编译为模块，操作系统在启动时并

不知道该接口的存在，需要用户在/etc/rc.d/目录中定义的初始启动脚本中写入命令或手动将模块插入内核空间来激活网络接口。这也给我们在何时加载网络设备驱动程序带来了灵活性。

### 9.2.4　网络设备驱动程序应用实例

以 ne2000 兼容网卡为例，来具体介绍基于模块的网络驱动程序的设计过程。可以参考文件 linux/drivers/net/ne.c 和 linux/drivers/net/8390.c。

#### 1．模块加载和卸载

ne2000 网卡的模块加载功能由 init_module()函数完成，具体过程及解释如下：

```
int init_module(void)
{
int this_dev, found = 0;
for (this_dev = 0; this_dev < MAX_NE_CARDS; this_dev++)
                                        //循环检测 ne2000 类型的网络设备接口
{
 struct net_device *dev = &dev_ne[this_dev];
                                        //获得网络接口对应的 net-device 结构指针
 dev->irq = irq[this_dev];              //初始化该接口的中断请求号
 dev->mem_end = bad[this_dev];          //初始化接收缓冲区的终点位置
 dev->base_addr = io[this_dev];         //初始化网络接口的 I/O 基地址
 dev->init = ne_probe;                  //初始化 init 为 ne_probe，后面介绍此函数
/*调用 registre_netdevice()向系统登记网络接口，在这个函数中将分配给网络接口在系统中唯一的名称。并且将该网络接口设备添加到系统管理的链表 dev-base 中进行管理。*/
if (register_netdev(dev) == 0) {
     found++;
     continue; }
...  //省略
}
return 0;}
```

模块卸载功能由 cleanup_module()函数，实现代码如下：

```
void cleanup_module(void)
{
int this_dev;
for (this_dev = 0; this_dev < MAX_NE_CARDS; this_dev++) {
                                                      //遍历整个 dev-net 数组
    struct net_device *dev = &dev_ne[this_dev]; //获得 net-device 结构指针
    if (dev->priv != NULL) {
        void *priv = dev->priv;
        struct pci_dev *idev = (struct pci_dev *)ei_status.priv;
if (idev)  idev->deactivate(idev);
                          //调用函数指针 idev->deactive 将已经激活的网卡关闭使用
free_irq(dev->irq, dev);
release_region(dev->base_addr, NE_IO_EXTENT);
                          //调用函数 release_region()释放该网卡占用的 I/O 地址空间
unregister_netdev(dev);//调用 unregister_netdev()注销这个 net_device()结构
kfree(priv);              //释放 priv 空间
     }
  }
}
```

### 2．网络接口初始化

实现此功能是由 ne_probe()函数完成的，前面已经提到过，在 init_module()函数中用它来初始化 init()函数指针。它主要对网卡进行检测，并且初始化系统中网络设备信息用于后面的网络数据的发送和接收。具体过程及解释如下：

```
int __init ne_probe(struct net_device *dev)
{
  unsigned int base_addr = dev->base_addr;
  /*初始化 dev-owner 成员，因为使用模块类型驱动，会将 dev-owner 指向对象 modules 结构
    指针。*/
  SET MODULE OWNER(dev);
  /*检测 dev->base_addr 是否合法，是则执行 ne-probe1()函数检测过程，如不是，则需要
    自动检测。*/
  if (base addr > 0x1ff)
    return ne probe1(dev, base addr);
  else if (base_addr != 0)
    return -ENXIO;
  /*如果有 ISAPnP 设备，则调用 ne_probe_isapnp()检测这种类型的网卡。*/
if (isapnp present() && (ne probe isapnp(dev) == 0))
    return 0;
 ...//省略
   return -ENODEV;
}
```

其中函数 ne_probe_isapnp()和 ne_probe19()的区别在于检测中断号上面。PCI 方式只需指定 I/O 基地址就可以自动获得 irq，是由 BIOS 自动分配的。但是 ISA 方式需要获得空闲的中断资源才能分配。

# 9.3　net_device 数据结构

struct net_device 结构体是整个网络驱动结构的核心，在本节中将专门介绍这个结构体。它定义了很多供网络协议接口层调用设备的标准方法，此结构是在 2.6 内核源码树文件中定义的，下面详细列出了其中主要的成员。

## 9.3.1　全局信息

结构体 net_device 的第一部分由下面成员组成：

```
char name[IFNAMSIZ]
```

设备名字：设备名字是由驱动设置的，包含一个%d 格式串，register_netdev 用一个数字替换它来形成一个唯一的设备名字；分配的编号从 0 开始，如 eth0 后面的 0。

```
unsigned long state
```

设备状态：这个成员包括几个标志。驱动正常情况下不直接操作这些标志；内核提供了一套实用函数。这些函数在我们进入驱动操作后将进行讨论。

```
struct net_device *next
```

全局列表中指向下一个设备的指针。驱动程序不应该对这个成员进行操作。

```
int (*init)(struct net_device *dev)
```

一个初始化函数。如果设置了这个指针，该函数被 register_netdev()调用完成对 net_device 结构的初始化。大部分现代的网络驱动不再使用该函数，初始化在注册接口前进行。

## 9.3.2　硬件信息

下面的成员包含了相对简单的设备低层硬件信息。它们是早期 Linux 网络的延续；大部分现代驱动确实使用它们（可能的例外是 if_port）。这里为了完整性，把它们列出。

```
unsigned long rmem end
unsigned long rmem_start
unsigned long mem_end
unsigned long mem_start
```

设备内存信息。这些成员保存设备使用的共享内存的开始和结束地址。如果设备有不同的接收和发送内存，mem 成员由发送内存使用，rmem 成员由接收内存使用。rmem 成员在驱动之外从不被引用。惯例上，设置 end 成员，所以 end - start 是可用的板上内存的数量。

```
unsigned long base_addr
```

这个成员表示网络接口的 I/O 基地址，在设备探测时赋值。ifconfig 目录可用来显示或修改当前值。base_addr 可以在系统启动时在内核命令行中显式赋值（通过 netdev=参数），或者在模块加载的时候赋值。这个成员内核也不使用它们。

```
unsigned char irq
```

表示中断号。Ifconfig 可以打印出 dev->irq 的值.。这个值通常在启动或者加载时设置并且在后来由 ifconfig 打印出来。

```
unsigned char if_port
```

在多端口设备中，这里表示使用的端口。例如这个成员用在同时支持同轴线（IF_PORT_10BASE2）和双绞线（IF_PORT_100BSAET）以太网连接。完整的已知端口类型设置定义在<linux/netdevie.h>中

```
unsigned char dma
```

为设备分配的 DMA 通道。这个成员只有在一些外设总线时有意义（如 ISA）它不在设备驱动之外使用。

## 9.3.3　接口信息

有关接口的大部分信息是由 ether_setup 函数设置的（或者任何其他对给定硬件类型适合的设置函数）。太网卡可以通过这个通用的函数设置大部分接口信息成员，要指出的是

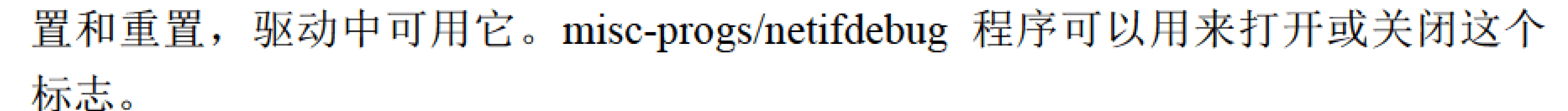
flags 和 dev_addr 成员是特定设备的，须在初始化时明确指定。

一些非以太网接口可以使用类似 ether_setup()函数。deviers/net/net_init.c 定义了一些类似的函数，包括下列内容。

- void ltalk_setup(struct net_device *dev)：用于设置一个 LocalTalk 设备的成员。
- void fc_setup(struct net_device *dev)：用于初始化光通道设备的成员。
- void fddi_setup(struct net_device *dev)：用于配置一个光纤分布数据接口（FDDI）网络的接口。
- void hippi_setup(struct net_device *dev)：用于预备给一个高性能并行接口（HIPPI）的高速互连驱动的成员。
- void tr_setup(struct net_device *dev)：用于处理令牌环网络接口的设置。

大多数设备会归于这些类别中的某一类。如果你使用的是其他的设备，则需要手工赋值以下的成员。

- unsigned short hard_header_len：指定硬件头部长度，就是被发送报文前面在 IP 头之前的字节数，或者其他协议信息。对于以太网接口 hard_header_len 的值是 14（ETH_HLEN）。
- unsigned mtu：表示最大传输单元（MTU）。这个成员在网络层用作驱动报文传输。以太网有一个 1500 字节的 MTU（ETH_DATA_LEN）。这个值可用 ifconfig 来改变。
- unsigned long tx_queue_len：在设备发送队列中可以排队的最大帧数。这个值由 ether_setup 设置为 1000，但是你可以修改它。例如 plip 使用 10 来避免浪费系统内存（相比真实以太网接口，plip 有一个低些的吞吐量）。
- unsigned short type：表示接口的硬件类型。这个 type 成员由 ARP 用来确定接口支持什么样的硬件地址。对于以太网接口正确的值是 ARPHRD_ETHER，这是由 ether_setup 设置的值。可认识的类型定义在<linux/if_arp.h>中。
- unsigned char addr_len、unsigned char broadcast[MAX_ADDR_LEN]和 unsigned char dev_addr[MAX_ADDR_LEN]：表示硬件（MAC）地址长度和设备硬件地址。以太网地址长度是 6 个字节（我们指的是接口板的硬件 ID），广播地址由 6 个 0xff 字节组成；ether_setup 安排成正确的值。设备地址，需要以特定于设备的方式从接口板读出，驱动应当将它复制到 dev_addr。硬件地址用来产生正确的以太网头，在报文传递给驱动发送之前。snull 设备不使用物理接口，它创造自己的硬件接口。
- unsigned short flags 和 int features：表示接口标志。flags 成员是一个位掩码，包括以下的位值：IFF_前缀代表“interface flags”。有些标志由内核管理，有些由接口在初始化时设置来表明接口的能力和其他特性。有效的标志在<linux/if.h>中有定义。
- IFF_UP：对于驱动来说这个标志是只读的。当接口激活并准备传送报文时内核打开它。
- IFF_BROADCAST：这个标志（由网络代码维护）说明接口允许广播。以太网板是这样。
- IFF_DEBUG：表示调试模式。这个标志用来控制你的 printk 调用的复杂性或者用于其他调试目的。尽管当前没有 in-tree 驱动使用这个标志，它可以通过 ioctl 来设



置和重置，驱动中可用它。misc-progs/netifdebug 程序可以用来打开或关闭这个标志。

- IFF_LOOPBACK：这个标志只在环回接口中设置。内核检查 IFF_LOOPBACK，以取代硬连线 lo 名子作为一个特殊接口。
- IFF_POINTOPOINT：这个标志说明接口连接到一个点对点链路。它由驱动设置或者由 ifconfig。例如 plip 和 PPP 驱动设置它。
- IFF_NOARP：这个表示接口不能进行 ARP。例如点对点接口不需要运行 ARP，它只能增加额外的流量却没有任何有用的信息。
- IFF_PROMISC：这个标志（由网络代码）用来激活混杂操作。默认情况下，以太网接口使用硬件过滤器来保证它们只接收广播报文和直接收到接口硬件地址的报文。报文嗅探器例如 tcpdump，在接口上设置混杂模式来存取在接口发送介质上经过的所有报文。
- IFF_MULTICAST：驱动设置这个标志来表示接口有组播发送能力。默认的情况下 ether_setup 设置 IFF_MULTICAST，如果你的驱动不支持组播，必须在初始化时清除这个标志。
- IFF_ALLMULTI：这个标志表示接口接收所有的组播报文。内核在主机进行组播路由时设置它，前提是 IFF_MULTICAST 置位。IFF_ALLMULTI 对驱动来说是只读的。
- IFF_MASTER 和 IFF_SLAVE：这些标志给负载均衡代码使用。接口驱动不需要使用它们。
- IFF_PORTSEL 和 IFF_AUTOMEDIA：这些标志表示设备可以在多个介质类型间切换；如无屏蔽双绞线（UTP）和同轴以太网电缆。如果设置了 IFF_AUTOMEDIA，设备会自动选择正确的介质。
- IFF_DYNAMIC：这个标志由驱动设置，表示接口的地址能够变化。目前内核没有使用它。
- IFF_RUNNING：这个标志表示接口已启动并在运行。它的存在大部分是因为和 BSD 兼容；内核很少用它。大多数网络驱动不需要关心 IFF_RUNNING。
- IFF_NOTRAILERS：在 Linux 中没有使用这个标志，是为了和 BSD 兼容才存在。当一个程序改变 IFF_UP 时，open()或者 stop()设备方法就被调用。进而，当 IFF_UP 或者其他标志修改了，set_multicast_lis()方法被调用。如果驱动需要进行某些动作来响应标志的修改，它必须在 set_multicast_list 中采取动作。如当 IFF_PROMISC 被置位或者复位，set_multicast_list 必须通知板上的硬件过滤器。

结构 net_device 的特性成员由驱动设置来告知，内核关于任何接口拥有的特别硬件能力。完整的集合是：

- NETIF_F_SG 和 NETIF_F_FRAGLIST：这两个标志控制发散/汇聚 I/O 的使用。如果接口可以发送一个报文，其由几个不同的内存段组成，应当设置 NETIF_F_SG。

**注意**：内核不对设备进行发散/汇聚 I/O 操作，如果它没有同时提供某些校验和形式。理由是如果内核不得不跨过一个分片的报文来计算校验和，它可能也复制数据并同时接合报文。

- NETIF_F_IP_CSUM、NETIF_F_NO_CSUM 和 NETIF_F_HW_CSUM：这些标志都是告知内核，不需要给一些或所有通过这个接口离开系统的报文进行校验。如果你的接口可以校验 IP 报文，就设置 NETIF_F_IP_CSUM。如果这个接口不曾要求校验和，就设置 NETIF_F_NO_CSUM。环回驱动设置了这个标志，snull 也设置，因为报文只通过系统内存传送，对它们来说没有机会（1 跳）被破坏，没有必要校验它们。如果你的硬件自己做校验，设置 NETIF_F_HW_CWSUM。
- NETIF_F_HIGHDMA：如果你的设备能够对高端内存进行 DMA。设置这个标志，没有这个标志，所有提供给你的驱动报文将在低端内存分配。
- NETIF_F_HW_VLAN_TX、NETIF_F_HW_VLAN_RX、NETIF_F_HW_VLAN_FILTER 和 NETIF_F_VLAN_CHALLENGED：这些选项描述你的硬件对 802.1qVLAN 报文的支持。VLAN 支持超出本章的内容。如果 VLAN 报文使你的设备混乱（其实不应该），设置标志 NETIF_F_VLAN_CHALLENGED。
- NETIF_F_TSO：如果你的设备能够进行 TCP 分段卸载，设置这个标志。TSO 是一个在这里不涉及的高级特性。

### 9.3.4　设备方法

和字符和块驱动一样，每个网络设备都声明能操作它的函数。本节列出能够对网络接口进行的操作。有些操作可以留作 NULL，其他的通常是不被触动的，因为 ether_setup 给它们安排了合适的方法。

网络接口的设备方法可分为 2 组：基本的和可选的。基本的方法包括那些必需的能够使用接口的；可选的方法实现更多高级的不是严格要求的功能。下列是基本方法：

```
int (*open)(struct net_device *dev);
```

打开接口：任何时候 ifconfig 激活它，接口被打开。Open()方法应当注册它需要的任何系统资源（I/O）口、IRQ、DMA 等），打开硬件，进行其他你的设备要求的设置。

```
int (*stop)(struct net_device *dev);
```

停止接口。接口停止当它被关闭。这个函数应当恢复在打开时进行的操作。

```
int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
```

起始报文的发送方法。完整的报文（协议头和所有）包含在一个 socket 缓存区（sk_buff）结构。socket 缓存在本章后面介绍。

```
int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned
short type, void *daddr, void *saddr, unsigned len);
```

用之前取到的源和目的硬件地址来建立硬件头的函数（在 hard_start_xmit 前调用）。它的工作是将作为参数传给它的信息组织成一个合适的特定于设备的硬件头。eth_header 是以太网类型接口的默认函数：ether_setup 针对性地对这个成员赋值。

```
int (*rebuild_header)(struct sk_buff *skb);
```

用来在 ARP 解析完成后但是在报文发送前重建硬件头的函数。以太网设备使用默认的

函数使用 ARP 支持代码来填充报文缺失的信息。

```
void (*tx_timeout)(struct net_device *dev);
```

一个报文发送没有在一个合理的时间内完成时，由网络代码调用的方法，可能是丢失一个中断或者接口被锁住。它应当处理这个问题并恢复报文发送。

```
struct net_device_stats *(*get_stats)(struct net_device *dev);
```

任何时候当一个应用程序需要获取接口的统计信息时，调用这个方法。当 ifconfig 或者 netstat -i 运行时：

```
int (*set_config)(struct net_device *dev, struct ifmap *map);
```

改变接口配置。这个方法是配置驱动的入口点。设备的 I/O 地址和中断号可以在运行时使用 set_config 来改变。这种能力可由系统管理员在接口没有探测到时使用。现代硬件正常的驱动一般不需要实现这个方法。

剩下的设备操作是可选的：

```
int weight;
int (*poll)(struct net_device *dev; int *quota);
```

由适应 NAPI 的驱动提供的方法，用来在查询模式下操作的接口，中断关闭着。

```
void (*poll_controller)(struct net_device *dev);
```

在中断关闭的情况下，要求驱动检查接口上的事件函数。它用于特殊的内核中的网络任务，例如远程控制台和使用网络的内核调试。

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

处理特定于接口的 ioctl 命令。如果接口不需要相应的 net_device 结构中的成员可留为 NULL，任何特定于接口的命令。

```
void (*set_multicast_list)(struct net_device *dev);
```

当设备的组播列表改变和当标志改变时调用的方法。

```
int (*set_mac_address)(struct net_device *dev, void *addr);
```

如果接口支持改变它的硬件地址的能力，则可以实现这个函数。很多接口根本不支持这个能力。其他的使用默认的 eth_mac_adr 实现（在 deivers/net/net_init.c）。eth_mac_addr 只复制新地址到 dev->dev_addr，只在接口没有运行时做这件事。使用 eth_mac_addr 的驱动应当在它们的 open()方法中从 dev->dev_addr 里设置硬件 MAC 地址。

```
int (*change_mtu)(struct net_device *dev, int new_mtu);
```

当接口的最大传输单元（MTU）改变时动作的函数。如果用户改变 MTU 时驱动需要做一些特殊的事情，它应当声明它自己的函数，否则，默认的会将事情做对。

```
int (*header_cache) (struct neighbour *neigh, struct hh_cache *hh);
```

header_cache 被调用来填充 hh_cache 结构，使用一个 ARP 请求的结果。几乎全部类似以太网的驱动可以使用默认的 eth_header_cache 实现。

```
int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev,
unsigned char *haddr);
```

在响应一个变化中，更新 hh_cache 结构中的目的地址的方法。以太网设备使用 eth_header_cache_update。

```
int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);
```

hard_header_parse 方法从包含在 skb 中的报文中抽取源地址，复制到 haddr 的缓存区。函数的返回值是地址的长度。以太网设备通常使用 eth_header_parse。

### 9.3.5　公用成员

结构 net_device 剩下的数据成员由接口使用来持有有用的状态信息。 有些是 ifconfig 和 netstat 用来提供给用户关于当前配置的信息。因此，接口应该给这些成员赋值：

```
unsigned long trans_start;
unsigned long last_rx;
```

保存一个 jiffy 值的成员。当开始发送和收到一个报文时，驱动负责分别更新这些值。trans_start 值被网络子系统用来探测发送器加锁。last_rx 目前没有用到，但是驱动应当尽量维护这个成员以备将来使用。

```
int watchdog_timeo;
```

网络层认为一个传送超时发生前应当过去的最小时间（按 jiffy 计算），调用驱动的 tx_timeout 函数。

```
void *priv;
```

filp->private_data 的对等者。在现代的驱动里，这个成员由 alloc_netdev 设置，不应当直接存取，使用 netdev_priv 代替。

```
struct dev_mc_list *mc_list;
int mc_count;
```

处理组播发送的成员。mc_count 是 mc_list 中的项数目。

```
spinlock_t xmit_lock;
int xmit_lock_owner;
```

xmit_lock 用来避免对驱动的 hard_start_xmit()函数多个同时调用。xmit_lock_owner 是已获得 xmit_lock 的 CPU 号。驱动应当不改变这些成员的值。

结构 net_device 中有其他的成员，但是网络驱动不需要用到它们。

## 9.4　DM9000 网卡概述

有些嵌入式处理器并没有集成以太网 MAC 层控制器，对于这种处理器，人们选择使用集成了 MAC 控制器和 PHY 层的以太网芯片，来扩展网络接口。DM9000 就是这样一种

常用的自适应以太网芯片。

### 9.4.1　DM9000 网卡总体介绍

DM9000 是一种快速以太网控制处理器，它合成了 MAC、PHY 和 MMU。该处理器配备有标准 10M/100M 自适应，16K 容量的 FIFO，4 路多功能 GPIO，掉电，全双工工作等功能。支持以太网接口协议。

网络数据有时是以猝发形式收到的，因此，DM9000 还集成了接收缓冲区，以便在接收到数据时能把数据放在这个缓冲区中，然后由数据链路层直接从这个缓冲区里取出数据。链路层通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡，它们共同处理与电缆的物理接口细节数据，其缓冲区可用来暂时存储要发送或接收的帧。

DM9000 还提供介质无关接口，用来连接所有提供支持介质无关接口功能的家用电话线网络设备或其他收发器。DM9000 支持 8 位、16 位和 32 位接口访问内部存储器，以支持不同的处理器。DM9000 物理协议层接口完全支持使用 10MBps 下 3 类、4 类、5 类非屏蔽双绞线和 100MBps 下 5 类非屏蔽双绞线，这完全符合 IEEE 802.3u 规格。它的自动协调功能可以自动完成配置以最大限度地适合其线路带宽，还支持 IEEE 802.3x 全双工流量控制。这个工作在 DM9000 里面是非常简单的，所以用户可以容易地移植任何系统下的网卡驱动程序。

### 9.4.2　DM9000 网卡的特点

DM9000 网卡具有如下特点。

- 支持处理器读写内部存储器的数据操作命令以字节/字/双字的长度进行；
- 集成 10/100M 自适应收发器；
- 支持介质无关接口；
- 支持背压模式半双工流量控制模式；
- IEEE 802.3x 流量控制的全双工模式；
- 支持唤醒帧，链路状态改变和远程的唤醒；
- 4K 双字 SRAM；
- 支持自动加载 EEPROM 里面生产商 ID 和产品 ID；
- 支持 4 个通用输入输出口；
- 超低功耗模式；
- 功率降低模式；
- 电源故障模式；
- 可选择 1：1 YL18-2050S,YT37-1107S 或 5∶4 变压比例的变压器降低格外功率；
- 兼容 3.3V 和 5.0V 输入输出电压；
- 100 脚 CMOS LQFP 封装工艺。

## 9.4.3　内部寄存器

DM9000 包含一系列可被访问的控制状态寄存器，这些寄存器是字节对齐的，它们在硬件或软件复位时被设置成初始值，以下为 DM9000 的寄存器功能详解。

### 1．网络控制寄存器（NCR）

网络控制寄存器用于对 DM9000 工作状态的控制，可以使 DM9000 复位，功能描述如表 9.1 所述。

表 9.1　网络控制寄存器（Network Control Register）

| 功　能 | 位 | 描　述 |
| --- | --- | --- |
| EXT_PHY | [7] | 1 选择外部 PHY，0 选择内部 PHY，不受软件复位影响 |
| WAKEEN | [6] | 事件唤醒使能，1 使能，0 禁止并清除事件唤醒状态，不受软件复位影响 |
| 保留 | [5] | |
| FCOL | [4] | 1 强制冲突模式，用于用户测试 |
| FDX | [3] | 全双工模式。内部 PHY 模式下只读，外部 PHY 下可读写 |
| LBK | [1-2] | 回环模式（Loopback）00 通常，01MAC 内部回环，10 内部 PHY 100M 模式数字回环，11 保留 |
| RST | [0] | 1 软件复位，10us 后自动清零 |

### 2．网络状态寄存器（NSR）

网络状态寄存器，通过该寄存器可以获知 DM9000 当前的工作状态，例如是否处于连接状态，发送数据是否完毕，是否处于睡眠状态等，功能描述如表 9.2 所述。

表 9.2　网络状态寄存器（Network Status Register）

| 功　能 | 位 | 描　述 |
| --- | --- | --- |
| SPEED | [7] | 媒介速度，在内部 PHY 模式下，0 为 100Mbps，1 为 10Mbps。当 LINKST=0 时，此位不用 |
| LINKST | [6] | 连接状态，在内部 PHY 模式下，0 为连接失败，1 为已连接 |
| WAKEST | [5] | 唤醒事件状态。读取或写 1 将清零该位。不受软件复位影响 |
| 保留 | [4] | |
| TX2END | [3] | TX（发送）数据包 2 完成标志，读取或写 1 将清零该位。数据包指针 2 传输完成 |
| TX2END | [2] | TX（发送）数据包 1 完成标志，读取或写 1 将清零该位。数据包指针 1 传输完成 |
| RXOV | [1] | RX（接收）FIFO（先进先出缓存）溢出标志 |
| 保留 | [0] | |

### 3．发送控制寄存器（TCR）

发送控制寄存器，可以控制发送使能，功能描述如表 9.3 所述。

表 9.3　发送控制寄存器（TX Control Register）

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| 保留 | [7] | |
| TJDIS | [6] | Jabber 传输使能。1 使能 Jabber 传输定时器（2048 字节），0 禁止 |
| EXCECM | [5] | 额外冲突模式控制。0 当额外的冲突计数多于 15 则终止本次数据包，1 始终尝试发送本次数据包 |
| PAD_DIS2 | [4] | 禁止为数据包指针 2 添加 PAD |
| CRC_DIS2 | [3] | 禁止为数据包指针 2 添加 CRC 校验 |
| PAD_DIS2 | [2] | 禁止为数据包指针 1 添加 PAD |
| CRC_DIS2 | [1] | 禁止为数据包指针 1 添加 CRC 校验 |
| TXREQ | [0] | TX（发送）请求。发送完成后自动清零该位 |

注释：Jabber 是一个有 CRC 错误的长帧（大于 1518byte 而小于 6000byte）或是数据包重组错误。原因：它可能导致网络丢包，多是由于网站有硬件或软件错误。

### 4．数据包指针1的发送状态寄存器1（TSR_I）

数据包指针 1 的发送状态寄存器 1 功能描述，如表 9.4 所述。

表 9.4　数据包指针 1 的发送状态寄存器 1（TX Status Register I）

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| TJTO | [7] | Jabber 传输超时。该位置位表示由于多于 2048 字节数据被传输而导致数据帧被截掉 |
| LC | [6] | 载波信号丢失。该位置位表示在帧传输时发生红载波信号丢失。在内部回环模式下该位无效 |
| NC | [5] | 无载波信号。该位置位表示在帧传输时无载波信号。在内部回环模式下该位无效 |
| LC | [4] | 冲突延迟。该位置位表示在 64 字节的冲突窗口后又发生冲突 |
| COL | [3] | 数据包冲突。该位置位表示传输过程中发生冲突 |
| EC | [2] | 额外冲突。该位置位表示由于发生了第 16 次冲突（即额外冲突）后，传送被终止 |
| 保留 | [1-0] | |

### 5．数据包指针2的发送状态寄存器2（TSR_II）

数据包指针 2 的发送状态寄存器 2 功能描述，如表 9.5 所述。

表 9.5　数据包指针 2 的发送状态寄存器 2（TX Status Register II）

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| TJTO | [7] | Jabber 传输超时。该位置位表示由于多于 2048 字节数据被传输而导致数据帧被截掉 |
| LC | [6] | 载波信号丢失。该位置位表示在帧传输时发生红载波信号丢失。在内部回环模式下该位无效 |

续表

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| NC | [5] | 无载波信号。该位置位表示在帧传输时无载波信号。在内部回环模式下该位无效 |
| LC | [4] | 冲突延迟。该位置位表示在 64 字节的冲突窗口后又发生冲突 |
| COL | [3] | 数据包冲突。该位置位表示传输过程中发生冲突 |
| EC | [2] | 额外冲突。该位置位表示由于发生了第 16 次冲突（即额外冲突）后，传送被终止 |
| 保留 | [1-0] | |

### 6．接收控制寄存器（RCR）

接收控制寄存器，可以控制接收使能，功能描述如表 9.6 所述。

**表 9.6　接收控制寄存器（RX Control Register）**

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| 保留 | [7] | |
| WTDIS | [6] | 看门狗定时器禁止。1 禁止，0 使能 |
| DIS_LONG | [5] | 丢弃长数据包。1 为丢弃数据包长度超过 1522 字节的数据包 |
| DIS_CRC | [4] | 丢弃 CRC 校验错误的数据包 |
| ALL | [3] | 忽略所有多点传送 |
| RUNT | [2] | 忽略不完整的数据包 |
| PRMSC | [1] | 混杂模式（Promiscuous Mode） |
| RXEN | [0] | 接收使能 |

### 7．接收状态寄存器（RSR）

接收状态寄存器，当有接收中断到来时，可以通过读取该寄存器，进一步了解当前 DM9000 网卡的接收状态，从而确定目前接受的这一帧数据应该如何处理，功能描述如表 9.7 所述。

**表 9.7　接收状态寄存器（RX Status Register）**

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| RF | [7] | 不完整数据帧。该位置位表示接收到小于 64 字节的帧 |
| MF | [6] | 多点传送帧。该位置位表示接收到帧包含多点传送地址 |
| LCS | [5] | 冲突延迟。该位置位表示在帧接收过程中发生冲突延迟 |
| RWTO | [4] | 接收看门狗定时溢出。该位置位表示接收到大于 2048 字节数据帧 |
| PLE | [3] | 物理层错误。该位置位表示在帧接收过程中发生物理层错误 |
| AE | [2] | 对齐错误（Alignment）。该位置位表示接收到的帧结尾处不是字节对齐，即不是以字节为边界对齐 |
| CE | [1] | CRC 校验错误。该位置位表示接收到的帧 CRC 校验错误 |
| FOE | [0] | 接收 FIFO 缓存溢出。该位置位表示在帧接收时发生 FIFO 溢出 |

### 8．接收/发送溢出控制寄存器（RTFCR）

接收/发送溢出控制寄存器功能描述，如表 9.8 所述。

表 9.8　接收/发送溢出控制寄存器（RX/TX Flow Control Register）

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| TXP0 | [7] | 1 发送暂停包。发送完成后自动清零，并设置 TX 暂停包时间为 0000H |
| TXPF | [6] | 1 发送暂停包。发送完成后自动清零，并设置 TX 暂停包时间为 FFFFH |
| TXPEN | [5] | 强制发送暂停包使能。按溢出门限最高值使能发送暂停包 |
| BKPA | [4] | 背压模式。该模式仅在半双工模式下有效。当接收 SRAM 超过 BPHW 并且接收新数据包时，产生一个拥挤状态 |
| BKPM | [3] | 背压模式。该模式仅在半双工模式下有效。当接收 SRAM 超过 BPHW 并数据包 DA 匹配时，产生一个拥挤状态 |
| RXPS | [2] | 接收暂停包状态。只读，清零时允许接收 |
| RXPCS | [1] | 接收暂停包当前状态 |
| FLCE | [0] | 溢出控制使能控。1 设置使能溢出制模式 |

### 9．传送数据长度寄存器

- DM_TXPLL（0xFC）：传送数据长度低字节寄存器，在发送数据时，该寄存器存放发送的数据长度的低字节。
- DM_TXPLH（0xFD）：传送数据长度高字节寄存器，在发送数据时，该寄存器存放发送的数据长度的高字节。

### 10．中断状态寄存器（ISR）

中断状态寄存器，当一个中断到来时，该寄存器存放着中断类型。DM9000 中断处理函数通过读取该寄存器，得到目前中断信息，从而能够正确调用相应的中断处理子程序。读取该中断状态寄存器之后，还需要将读取结果存放回该寄存器，也就是需要清楚中断状态，否则将无法再次响应中断，功能描述如表 9.9 所述。

表 9.9　中断状态寄存器（Interrupt Status Register）

| 功　　能 | 位 | 描　　述 |
|---|---|---|
| IOMODE | [7-6] | 处理器模式。00 为 16 位模式，01 为 32 位模式，10 为 8 位模式，00 保留 |
| LNKCHG | [5] | 连接状态改变 |
| UDRUN | [4] | 传输“Underrun” |
| ROOS | [3] | 接收溢出计数器溢出 |
| ROS | [2] | 接收溢出 |
| PTS | [1] | 数据包传输 |
| PRS | [0] | 数据包接收 |

### 11．中断掩码寄存器（IMR）

中断掩码寄存器，该寄存器存放当前 DM9000 使能的中断类型。在该系统中，只让接

收中断使能。利用该寄存器，可以灵活地使得 DM9000 屏蔽中断，或者开启中断。例如在发送数据开始时，可以屏蔽中断，在发送结束后，再开启中断，这样可以使得 DM9000 工作的稳定性大大提高，功能描述如表 9.10 所述。

表 9.10　中断掩码寄存器（Interrupt Mask Registe）

| 功　能 | 位 | 描　述 |
|---|---|---|
| PAR | [7] | 1 使能指针自动跳回。当 SRAM 的读、写指针超过 SRAM 的大小时，指针自动跳回起始位置。需要驱动程序设置该位，若设置则 REG_F5（MDRAH）将自动为 0CH |
| 保留 | [6] | |
| LNKCHGI | [5] | 1 使能连接状态改变中断 |
| UDRUNI | [4] | 1 使能传输“Underrun”中断 |
| ROOI | [3] | 1 使能接收溢出计数器溢出中断 |
| ROI | [2] | 1 使能接收溢出中断 |
| PTI | [1] | 1 使能数据包传输终端 |
| PRI | [0] | 1 使能数据包接收中断 |

以上为 DM9000（A）常用寄存器功能的详细介绍，通过对这些寄存器的操作访问，可以实现对 DM9000 的初始化、数据发送、接收等相关操作。而要实现 ARP、IP、TCP 等功能，则需要对相关协议的理解，由编写相关协议或移植协议栈来实现。

## 9.4.4　功能描述

### 1．总线

总线是 ISA 总线兼容模式，8 个 I/O 基址，分别是 300H、310H、320H、330H、340H、350H、360H、370H。I/O 基址与设定引脚或内部 EEPROM 的共同选定。

访问芯片有两个地址端口，分别是地址端口和数据端口。当引脚 CMD 接地时，为地址端口；当引脚 CMD 接高电平时，为数据端口。在访问任何寄存器前，地址端口输入的是数据端口的寄存器地址，寄存器的地址必须保存在地址端口。

### 2．存储器直接访问控制

DM9000 提供 DMA（直接存取技术）来简化对内部存储器的访问。在对内部存储器起始地址完成编程后，然后发出伪读写命令就可以加载当期数据到内部数据缓冲区，可以通过读写命令寄存器来定位内部存储区地址。根据当前总线模式的字长使存储地址自动加 1，下一个地址数据将会自动加载到内部数据缓冲区。要注意的是在连续突发式的第一次访问时读写命令的内容。

内部存储器空间大少 16K 字节。低 3K 字节单元用作发送包的缓冲区，其他 13K 字节用作接收包的缓冲区。所以在写发送包存储器的时候，当存储器地址越界后，自动跳回 0 地址并置位 IMR 第 7 位。同样在读接收包存储器的时候，当存储器地址越界后，自动跳回起始地址 0x0c00。

### 3．包的发送

有两个指针，顺序命名为指针 1 和指针 2，能同时存储在发送包缓冲区。发送控制寄

存器（02H）控制冗余校验码和填充的插入，其状态分别记录在发送状态寄存器 1（03H）和发送状态寄存器 2（04H）发送器的起始地址是 0x00H，软件或硬件复位后默认是指针 1，先通过 DMA 端口写数据到发送包缓冲区，然后写字节计数长度到字节计数寄存器。

# 9.5　DM9000 网卡驱动程序移植

Linux 内核中已经有 DM9000 网卡驱动，源码文件为 drivers/net/dm9000.c。与前面几章移植类似，所要做的工作也是告诉内核 DM9000 芯片所使用的资源（访问地址、中断号等），使得这些资源可用。本节主要分析内核源码中的 dm9000.c 文件。

## 9.5.1　DM9000 网卡连接

由于必须告知内核 DM9000 芯片所使用的硬件资源，所以移植的首要任务是分析 DM9000 芯片的硬件连接情况，以获得访问地址、中断号等硬件资源。DM9000 芯片在开发板上的连接情况如图 9.9 所示。

从连接图可以确定：

（1）由于用 nGCS4 作为片选信号，所以访问 DM9000 的基址是 0X20000000，这是物理地址。

（2）地址线只有一条，即 ADDR2。这是由 DM9000 的特性决定的，DM9000 的地址信号和数据信号是复用的，使用 CMD 引脚来区分它们，CMD 为低电平时，数据总路线上传输的是地址信号，CMD 为高电平时数据总线上传输的是数据信号。访问 DM9000 内部寄存器时，需要先将 CMD 置为低电平，发出地址信号，然后将 CMD 置为高电平，读写数据。

（3）总路线位宽为 16 位，用 nWAIT 信号。

（4）用 EINT7 外部中断作为中断引脚。

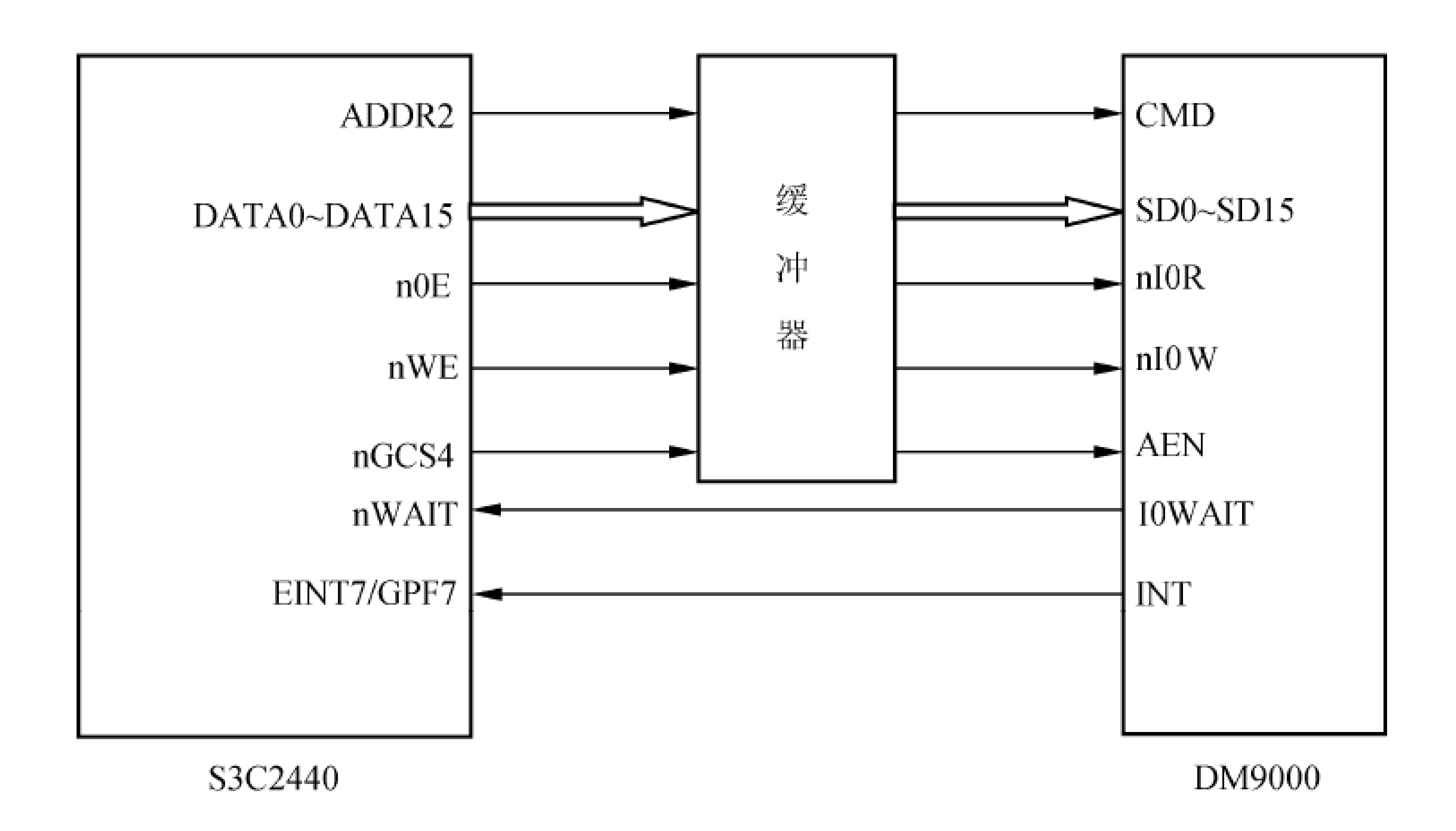



图 9.9　DM9000 网卡与芯片连接图

对源码的移植必须建立在读懂源码的基础上，所以在介绍驱动移植之前，必须先对内核 DM9000 网卡驱动程序进行分析。

## 9.5.2　驱动分析——硬件的数据结构

在内核源码中用 board_info 结构体来描述具体的硬件，它保存了一些硬件资源，其定义在 driver/net/dm9000.c 中。

```
typedef struct board_info {
    void __iomem *io_addr;  //地址基址，这两个地址是内核虚拟地址，不是物理地址
    void __iomem *io_data;  //数据基址
    u16 irq;                //中断号

    u16 tx_pkt_cnt;         //发包计数
    u16 queue pkt len;      //队列长度
    u16 queue_start_addr;   //队列开始地址
    u16 dbug_cnt;
    u8 io_mode;      /*0:word, 2:byte*/
    u8 phy_addr;
    unsigned int flags;
    unsigned int in suspend :1;
/*下面几个和 include/linux/dm9000.h 中定义的 struct dm9000_plat_data 一样，这个
源码从某种程度上说有点重复定义的嫌疑*/
    int debug_level;                                //调试级别

    void (*inblk)(void __iomem *port, void *data, int length);
                                                    //输入方法
    void (*outblk)(void __iomem *port, void *data, int length);
                                                    //输出方法
    void (*dumpblk)(void __iomem *port, int length);

    struct device   *dev;           // parent device 指向 platform_device->
    device

    struct resource *addr_res;      //找到在 devs.c 中定义的资源
    struct resource *data_res;

    struct resource *addr_req;      /*根据 addr_res 申请到的资源，上面两个只是知
    道了网卡所使用的资源，但是在内核中，物理资源的使用是要经过申请的。*/
    struct resource *data_req;
    struct resource *irq_res;

    struct mutex    addr_lock; /*phy and eeprom access lock*/

    spinlock_t lock;            //自旋锁

    struct mii_if_info mii;     // mii 信息
    u32 msg_enable;             //使能标志
} board_info_t;
```

这是一个驱动开发者自定义的结构体，用于保存该设备的相关信息（设备的属性如统计信息、读写操作、占用的 I/O 地址资源、状态）和相关操作，这些属性和操作是该设备的物理抽象。这个结构体最后被挂到了 net_device 的 priv 成员上，这就是所谓的网络设备

私有成员结构。

### 9.5.3　驱动分析——数据读写函数

由于 DM9000 的地址线和数据线是复用的，所以它有自己特别的读写方法，当要从网卡中某个寄存器读取值时，要先在地址基址中写入要读取的寄存器地址，然后再从数据基址中读出数据，方法如下：

```
static u8
ior(board_info_t * db, int reg)
{
    writeb(reg, db->io_addr);
    return readb(db->io_data);
}
```

当要从网卡中的某个寄存器写入值时，要先在地址基址中写入要写入的寄存器地址，然后再从数据基址中写入要写进去的数据，方法如下：

```
static void
iow(board_info_t * db, int reg, int value)
{
    writeb(reg, db->io_addr);
    writeb(value, db->io_data);
}
```

### 9.5.4　驱动分析——重置网卡

网卡启动前要先对网卡进行重置位，置位方法是向前面所介绍的网络控制寄存器（NCR）的置位位中写入 1，重置函数如下：

```
static void
dm9000_reset(board_info_t * db)
{
    dev_dbg(db->dev, "resetting device\n");    //调试信息

    /*RESET device*/
    writeb(DM9000_NCR, db->io_addr);//写入要操作的寄存器地址,这里是 DM9000_NCR
    udelay(200);                        //作一下延时
    writeb(NCR_RST, db->io_data);   //写入置位值，这里是 1
    udelay(200);
}
```

### 9.5.5　驱动分析——初始化网卡

DM9000 网卡的初始化工作主要由 driver/net/dm9000.c 中的 dm9000_probe 来完成，该函数完成的主要工作是获取和申请硬件资源、申请中断号、初始化 net_device 结构体，最后注册网络设备。这个函数有些长，下面只列出主要代码，读者可自己对照源码进行学习。

```
static int
dm9000_probe(struct platform_device *pdev)
```

```
{
    struct dm9000_plat_data *pdata = pdev->dev.platform_data; /*从传进来的
    平台设备结构中取出设备的 platform_data，在 devs.c 中有定义*/
    struct board_info *db;  /*用于指向网卡的私有数据，这也是这个函数要完成的功能，
    下面操作都是为了从传进来的参数 pdev 中探索并获取这个结构的相关信息，pdev->dev.
    platform data 中包函了设备的私有信息，一般在 DEV.C 中定义用于描述这个设备，在这
    个驱动中 pdev->dev.platform_data 没什么用处,因为这里的所有设备信息都是从资源地
    址探索到的*/
    struct net_device *ndev;                        //网络设备结构体
    const unsigned char *mac_src;
    unsigned long base;
    int ret = 0;
    int iosize;
    int i;
    u32 id val;

    /*获取*/
    ndev = alloc_etherdev(sizeof (struct board_info));/*分配并初始化一个
    netdevice，传入的参数是给 net_device->pritate 用的，net_device->pritate 将
    指向这个参数*/
    if (!ndev) {
        dev_err(&pdev->dev, "could not allocate device.\n");
        return -ENOMEM;
    }

    SET NETDEV DEV(ndev, &pdev->dev);
                    //设置设备的继承关系，ndev ->dev.parent = pdev->dev

    dev_dbg(&pdev->dev, "dm9000_probe()");

    /*setup board info structure*/
    db = (struct board_info *) ndev->priv;
    memset(db, 0, sizeof (*db));//清零网卡私有结构，下面将对里面的成员进行填充

    db->dev = &pdev->dev;       //父设备

    spin_lock_init(&db->lock); //初始化自旋锁
    mutex_init(&db->addr_lock);

    if (pdev->num resources < 2) {
        /*资源数是否大于等于 2，因为网卡要用到内存资源和中断资源共享至少两种资源。*/
        ret = -ENODEV;
        goto out;
    } else if (pdev->num resources == 2) {
        base = pdev->resource[0].start;

        if (!request_mem_region(base, 4, ndev->name)) {
                        //申请内存资源，资源必须申请后才可用
            ret = -EBUSY;
            goto out;
        }

        ndev->base_addr = base;     //填写基址，这里用的只是物理地址
        ndev->irq = pdev->resource[1].start;        //填写中断资源
        db->io_addr = (void __iomem *)base;         //地址基址
        db->io_data = (void __iomem *)(base + 4);  //数据基址
```

```
        /*ensure at least we have a default set of IO routines*/
        dm9000_set_io(db, 2);//设置网卡的默认 I/O 函数，后面将根据实际情况再设置

    } else {
        db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
                                                   //获取设备资源
        db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
        db->irq_res  = platform_get_resource(pdev, IORESOURCE_IRQ, 0);

        if (db->addr_res == NULL || db->data_res == NULL ||
            db->irq_res == NULL) {
            dev_err(db->dev, "insufficient resources\n");
            ret = -ENOENT;
            goto out;
        }

        i = res_size(db->addr_res);             //求地址端口资源大小，这里是 4
        db->addr_req = request_mem_region(db->addr_res->start, i,
                          pdev->name);          //申请 I/O 资源

        if (db->addr_req == NULL) {
            dev_err(db->dev, "cannot claim address reg area\n");
            ret = -EIO;
            goto out;
        }

        db->io_addr = ioremap(db->addr_res->start, i); /*把申请到的 I/O 资源
        （物理地址）映射到内核并保存映射地址*/
        if (db->io_addr == NULL) {
            dev_err(db->dev, "failed to ioremap address reg\n");
            ret = -EINVAL;
            goto out;
        }

        iosize = res_size(db->data_res);    //这里的返回值作为 DM9000 数据位宽
        db->data_req = request_mem_region(db->data_res->start, iosize,
                          pdev->name);      //申请数据接口资源

        if (db->data_req == NULL) {
            dev_err(db->dev, "cannot claim data reg area\n");
            ret = -EIO;
            goto out;
        }

        db->io_data = ioremap(db->data_res->start, iosize);
                                            //映射数据接口为虚拟地址

        if (db->io_data == NULL) {
            dev_err(db->dev,"failed to ioremap data reg\n");
            ret = -EINVAL;
            goto out;
        }

        /*fill in parameters for net-dev structure*/

        ndev->base_addr = (unsigned long)db->io_addr;
        ndev->irq    = db->irq_res->start;
                /*这里不申请中断线，要在打开时才申请，这样就不会一直占用中断线*/
```

```
        /*ensure at least we have a default set of IO routines*/
        dm9000 set io(db, iosize);
                //根据 DM9000 数据位宽设置读写数据帧的函数指针
    }

    /*check to see if anything is being over-ridden*/
    if (pdata != NULL) {
        /*check to see if the driver wants to over-ride the
        /*default IO width*/
        /*如果在 devs.c 文件中定义了 platform_device 结构中的 platform_data，则要
        重载这些方法和重设数据位宽，也就是在内核有两个地方可以设置这些数据，一个是 pl-
        atform_data，一个是 platform_device 中的 resource ，在移置时可以只设置一
        个地方*/
        if (pdata->flags & DM9000 PLATF 8BITONLY)
            dm9000_set_io(db, 1);          //根据平台数据重设数据位宽

        if (pdata->flags & DM9000_PLATF_16BITONLY)
            dm9000_set_io(db, 2);

        if (pdata->flags & DM9000_PLATF_32BITONLY)
            dm9000_set_io(db, 4);

        /*check to see if there are any IO routine
         * over-rides*/

        if (pdata->inblk != NULL)
            db->inblk = pdata->inblk;//用 devs.c 定义的 platform_device 结构
            中的 platform data 重载这些方法

        if (pdata->outblk != NULL)
            db->outblk = pdata->outblk;

        if (pdata->dumpblk != NULL)
            db->dumpblk = pdata->dumpblk;

        db->flags = pdata->flags;
    }


    dm9000_reset(db);   //现在可以复位芯片了：dm9000_reset(db); db 中已经包含了
    详细的芯片信息*/

    /*try two times, DM9000 sometimes gets the first read wrong*/

    for (i = 0; i < 8; i++) {          //读取芯片 ID 号
        id_val  = ior(db, DM9000_VIDL);
        id_val |= (u32)ior(db, DM9000 VIDH) << 8;
        id_val |= (u32)ior(db, DM9000_PIDL) << 16;
        id_val |= (u32)ior(db, DM9000_PIDH) << 24;

        if (id_val == DM9000_ID)        //判断是否为 0x90000A46
            break;
        dev_err(db->dev, "read wrong id 0x%08x\n", id_val);
    }

    if (id_val != DM9000_ID) {
        dev_err(db->dev, "wrong id: 0x%08x\n", id_val);
        ret = -ENODEV;
```

```
        goto out;
    }

    /*from this point we assume that we have found a DM9000*/

    /*driver system function*/
    ether_setup(ndev);
        //初始化以太网 ndev ，这里设置了以太网的一些通用的（和硬件无关的）参数和方法

    /*设置 ndev 的基本操作*/
    ndev->open          = &dm9000_open;
    ndev->hard_start_xmit       = &dm9000_start_xmit;
    ndev->tx_timeout            = &dm9000_timeout;
    ndev->watchdog_timeo        = msecs_to_jiffies(watchdog);
    ndev->stop                  = &dm9000_stop;
    ndev->set_multicast_list    = &dm9000_hash_table;
    ndev->ethtool_ops           = &dm9000_ethtool_ops;
    ndev->do_ioctl              = &dm9000_ioctl;

#ifdef CONFIG_NET_POLL_CONTROLLER
    ndev->poll_controller    = &dm9000_poll_controller;
#endif

    db->msg_enable          = NETIF_MSG_LINK;
    db->mii.phy_id_mask     = 0x1f;
    db->mii.reg_num_mask    = 0x1f;
    db->mii.force_media     = 0;
    db->mii.full_duplex     = 0;
    db->mii.dev             = ndev;
    db->mii.mdio_read       = dm9000_phy_read;
    db->mii.mdio_write      = dm9000_phy_write;

    mac_src = "eeprom";

    /*try reading the node address from the attached EEPROM*/
    for (i = 0; i < 6; i += 2)
        dm9000_read_eeprom(db, i / 2, ndev->dev_addr+i);

    if (!is_valid_ether_addr(ndev->dev_addr)) {
        /*try reading from mac*/

        mac_src = "chip";
        for (i = 0; i < 6; i++)
            ndev->dev_addr[i] = ior(db, i+DM9000_PAR);
    }

    if (!is_valid_ether_addr(ndev->dev_addr))
        dev_warn(db->dev, "%s: Invalid ethernet MAC address. Please "
             "set using ifconfig\n", ndev->name);

    /*将 ndev 记录于平台设备 platform_dev 中，注册 ndev。这也是 ndev 与 platform_dev
    建立联系的地方。可以这么理解，Linux 的设备模型负责的只是设备的管理（检测、启动、移
    除），而如何访问这个设备的数据，比如说以字符流模式，块设备方式，网络接口，则定义相
    应的 cdev、gendisk、ndev，然后注册到内核。所有的数据访问工作都以这 3 种界面提供。
    这里的 ndev 在其他驱动中将换成其他自定义的结构，这里是因为 ndev 包含了 bd*/
    platform_set_drvdata(pdev, ndev);//pdev->dev->driver_data=ndev
    ret = register_netdev(ndev);

    if (ret == 0) {
        DECLARE_MAC_BUF(mac);
```

```
        printk("%s: dm9000 at %p,%p IRQ %d MAC: %s (%s)\n",
              ndev->name,  db->io_addr, db->io_data, ndev->irq,
              print mac(mac, ndev->dev addr), mac src);
    }
    return 0;

out:
    dev_err(db->dev, "not found (%d).\n", ret);

    dm9000_release_board(pdev, db);
    free_netdev(ndev);

    return ret;
}
```

## 9.5.6　驱动分析——打开和关闭网卡

打开网卡就是激活网络接口，使它能接收来自网络的数据并且传送到网络协议栈的上面，也可以将数据发送到网络上，而网卡的关闭就是使网络接口停止工作。在这里只分析打开函数，关闭函数留给读者自行分析，DM9000 网卡的打开函数分析如下：

```
static int
dm9000_open(struct net_device *dev)
{
    board_info_t *db = (board_info_t *) dev->priv;     //获取网卡数据结构
    unsigned long irqflags = db->irq res->flags & IRQF TRIGGER MASK;

    if (netif_msg_ifup(db))                              //设置使能标志
        dev_dbg(db->dev, "enabling %s\n", dev->name);
    if (irqflags == IRQF_TRIGGER_NONE) {
        dev warn(db->dev, "WARNING: no IRQ resource flags set.\n");
        irqflags = DEFAULT_TRIGGER;
    }

    irqflags |= IRQF_SHARED;                             //设置中断共享

   if (request_irq(dev->irq, &dm9000_interrupt, irqflags, dev->name, dev))
    /*申请中断线，要在打开时才申请，这样就不会一直占用中断线*/
        return -EAGAIN;

    /*Initialize DM9000 board*/
    dm9000_reset(db);        //重置网卡，这个函数在前面讲过了
    dm9000 init dm9000(dev);
                             //初始化网卡，这里主要是对 DM9000 网卡的寄存器进行设置

    /*Init driver variable*/
    db->dbug cnt = 0;

    mii_check_media(&db->mii, netif_msg_link(db), 1);
    netif_start_queue(dev);/*netif_start_queue 用来告诉上层网络协定这个驱动
    程序还有空的缓冲区可用，请把下一个封包送进来。*/

    return 0;
}
```

## 9.5.7　驱动分析——数据包的发送与接收

在驱动程序层次上，数据包的发送和接收都是通过底层对硬件的读写来完成的。当网络上的数据到来时，将触发硬件中断，根据注册的中断向量表确定处理函数，进入中断向量处理函数，将数据送到上层协议进行处理或者转发出去。

当协议层已经封装好上层协议数据的 skb_buffer 后，将调用 dm9000_start_xmit（struct sk_buff *skb, struct net_device *dev）函数把数据发出，数据的发送函数分析如下：

```
static int
dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    unsigned long flags;
    board_info_t *db = (board_info_t *) dev->priv;      //获取网卡数据

    dm9000_dbg(db, 3, "%s:\n", __func__);

    if (db->tx_pkt_cnt > 1)                             //网卡正忙，反回 1
        return 1;

    spin_lock_irqsave(&db->lock, flags);                //获取自旋锁

    /*Move data to DM9000 TX RAM*/
    writeb(DM9000_MWCMD, db->io_addr);      //存储器读地址自动增加的读数据命令

    (db->outblk)(db->io_data, skb->data, skb->len);     //把数据发送出去
    dev->stats.tx_bytes += skb->len;                    //统计发送字节数

    db->tx_pkt_cnt++;                       //正在发送中的数据计数加 1
    /*TX control: First packet immediately send, second packet queue*/
    /*如果只有当前包发送，写指令，写数据帧，发送包。如果多于一个数据包正在发送，当前帧不发送。*/
    if (db->tx_pkt_cnt == 1) {
        /*Set TX length to DM9000*/
        iow(db, DM9000_TXPLL, skb->len);
        iow(db, DM9000_TXPLH, skb->len >> 8);

        /*Issue TX polling command*/
        iow(db, DM9000_TCR, TCR_TXREQ);     /*Cleared after TX complete*/

        dev->trans_start = jiffies;/*save the time stamp*/
    } else {
        /*Second packet*/
        db->queue_pkt_len = skb->len;
        netif_stop_queue(dev);                      //网卡停止接收数据
    }

    spin_unlock_irqrestore(&db->lock, flags);   //取消自旋

    /*free this SKB*/
    dev_kfree_skb(skb);                             //执行释放内存空间的动作

    return 0;
}
```

网络数据包到达，DM9000 自动接收并存放在 DM9000 内部 RAM 中，产生中断。在中断处理中识别中断原因并调用接收处理函数，接收函数分析如下：

```
static void
dm9000_rx(struct net_device *dev)
{
    board_info_t *db = (board_info_t *) dev->priv;
    struct dm9000 rxhdr rxhdr;
    struct sk_buff *skb;
    u8 rxbyte, *rdptr;
    bool GoodPacket;
    int RxLen;

    /*Check packet ready or not*/
    do {
        /*读取芯片相关寄存器，确认 DM9000 正确的收到一帧数据。*/
        ior(db, DM9000_MRCMDX);/*Dummy read*/

        /*Get most updated data*/
        rxbyte = readb(db->io data);

        /*Status check: this byte must be 0 or 1*/
        if (rxbyte > DM9000 PKT RDY) {
            dev_warn(db->dev, "status check fail: %d\n", rxbyte);
            iow(db, DM9000_RCR, 0x00); /*Stop Device*/
            iow(db, DM9000_ISR, IMR_PAR);  /*Stop INT request*/
            return;
        }

        if (rxbyte != DM9000 PKT RDY)
            return;

        /*A packet ready now  & Get status/length*/
        GoodPacket = true;
        writeb(DM9000 MRCMD, db->io addr);

        (db->inblk)(db->io_data, &rxhdr, sizeof(rxhdr));

        RxLen = le16_to_cpu(rxhdr.RxLen);

        if (netif msg rx status(db))
            dev_dbg(db->dev, "RX: status %02x, length %04x\n",
                rxhdr.RxStatus, RxLen);

        /*Packet Status check*/
        if (RxLen < 0x40) {
            GoodPacket = false;
            if (netif msg rx err(db))
                dev_dbg(db->dev, "RX: Bad Packet (runt)\n");
        }

        if (RxLen > DM9000_PKT_MAX) {
            dev dbg(db->dev, "RST: RX Len:%x\n", RxLen);
        }

        if (rxhdr.RxStatus & 0xbf) {
```

```
            GoodPacket = false;
            if (rxhdr.RxStatus & 0x01) {
                if (netif msg rx err(db))
                    dev_dbg(db->dev, "fifo error\n");
                dev->stats.rx fifo errors++;
            }
            if (rxhdr.RxStatus & 0x02) {
                if (netif msg rx err(db))
                    dev_dbg(db->dev, "crc error\n");
                dev->stats.rx_crc_errors++;
            }
            if (rxhdr.RxStatus & 0x80) {
                if (netif msg rx err(db))
                    dev_dbg(db->dev, "length error\n");
                dev->stats.rx length errors++;
            }
        }

        /*Move data from DM9000*/
        /*申请 skb_buffer，将数据从 DM9000 中复制到 skb_buffer 中*/
        if (GoodPacket
            && ((skb = dev alloc skb(RxLen + 4)) != NULL)) {
            skb reserve(skb, 2);
            rdptr = (u8 *) skb_put(skb, RxLen - 4);

            /*Read received packet from RX SRAM*/

            (db->inblk)(db->io data, rdptr, RxLen);
            dev->stats.rx_bytes += RxLen;

            /*Pass to upper layer*/
            skb->protocol = eth type trans(skb, dev);      //去除以太网头
            netif_rx(skb);                         //把 skb_buffer 交给上层协议
            dev->stats.rx_packets++;

        } else {
            /*need to dump the packet's data*/

            (db->dumpblk)(db->io_data, RxLen);
        }
    } while (rxbyte == DM9000_PKT_RDY);
}
```

## 9.5.8　DM9000 网卡驱动程序移植

读懂了 DM9000 网卡驱动程序源码后，就可以开始移植这个驱动了，具体方法如下。

### 1．定义网卡设备

硬件的使用需要知道硬件所用到的资源，如 I/O 端口和中断号等，在 arch/arm/plat-s3c24xx 的 devs.c 中添加 DM9000 用到的地址端口、数据端口和中断号，这些都要在了解了硬件连接后才知道用到什么资源，读者可以回头去阅读相关的硬件连接图，代码如下：

```
static struct resource s3c_dm9000_resource[] = {
        [0] = {
          .start = 0x20000000,   //对应电路图 nGCS4
          .end   = 0x20000003,
          .flags = IORESOURCE_MEM,
       },
        [1] = {
          .start = 0x20000004,   //对应 ADDR2
          .end   = 0x20000007,   //   0x3f
          .flags = IORESOURCE_MEM,
       },
        [2] = {
          .start = IRQ_EINT7,    //对应图中的 EINT7
          .end   = IRQ_EINT7,
          .flags = IORESOURCE_IRQ,
       }
};
```

添加 DM9000 平台数据，该数据用于内核传递给驱动程序，这个结构体在 arch/arm/mach-s3c2410/devs.c 定义 platform_device 时，将被挂载到 struct platform_device 结构成员 dev.platform_data 中。DM9000 平台数据结构在 include/linux/dm9000.h 中的定义如下：

```
struct dm9000_plat_data {
    unsigned int    flags;

    /*allow replacement IO routines*/

    void    (*inblk)(void __iomem *reg, void *data, int len);
    void    (*outblk)(void __iomem *reg, void *data, int len);
    void    (*dumpblk)(void __iomem *reg, int len);
};
```

在 arch/arm/plat-s3c24xx 的 devs.c 中，只添加以下代码便可以了。

```
static struct dm9000_plat_data s3c_device_dm9000_platdata = {
        .flags= DM9000_PLATF_16BITONLY
};
struct platform_device s3c_device_dm9000 = {
   .name= "dm9000",                                   //设备名字
   .id= -1,                                           //设备 ID 让内核自动编号
   .num_resources= ARRAY_SIZE(s3c_dm9000_resource),//用到的资源数
   .resource= s3c_dm9000_resource,
                                       //用到的资源，在前面已经定义了，这里引用它
   .dev= {
.platform_data = &s3c_device_dm9000_platdata,         //引用平台数据
   }
};
EXPORT_SYMBOL(s3c_device_dm9000);  //让其他文件可以引用到这里定义的变量
/**********whs add 2009-3-28 12:06------end ***************/
```

## 2．添加变量声名

在前面步骤中，定义了 s3c_device_dm9000 并用 EXPORT_SYMBOL(s3c_device_dm9000)使之变为全局变量，所以在这里要添加它的声明，在 arch/arm/plat-S3C24xx/include/

plat/devs.hc 添加声明如下：

```
extern struct platform_device s3c_device_dm9000;
```

### 3．添加平台设备列表

内核中用 smdk2440_devices 初始化列表保存系统启动时要初始化的设备，所以在这里要把网卡设备加进去，在 arch/arm/mach-s3c2440/mach-smdk2440.c 中将 s3c_device_dm9000 添加到平台设备列表中。

```
static struct platform_device *smdk2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,
        &s3c_device_dm9000,/***whs add 2009-3-28 12:06****/
};
```

### 4．修改dm9000.c

经过上述步骤，DM9000 网卡设备就已经成功注册进驱动核心。之后还需要做两方面的工作：设置芯片 MAC 地址，使能 DM9000 的中断，下面对 drivers/net/dm9000.c 中初始化函数进行修改。

根据 2440 资料，有几个地方需要设置：

（1）设置 GPGCON 使 GPG1 功能设置为 EINT7，可以用以下函数实现：

```
s3c2410_gpio_cfgpin(S3C2410_GPG1, S3C2410_GPF3_EINT7);
```

（2）外部中断 EXTINT1 的[6:4]位置 100 上升沿触发中。

因为要用到一些 GPIO 寄存器地址，所以必须在文件的开头把相关的 mach/regs-gpio.h 文件包含进来，在 dm9000. c 的开始添加如下定义：

```
#include <mach/regs-gpio.h>
static void  *extint1;                      //中断寄存器地址
static void  *intmsk;                       //中断屏蔽寄存器地址
#define EINTMASK         (0x560000a4)       //外部中断屏蔽
#define EXTINT1          (0x5600008c)       //外部中断方式
#define INTMSK           (0x4A000008)       //中断屏蔽
```

现在开始设置芯片的 MAC 地址和使能 DM9000 的中断，在 dm9000. c 中初始化函数 dm9000_probe()的 register_netdev 前添加如下代码：

```
memcpy(ndev->dev_addr,"\0andy1",6);                         //MAC 地址
extint1=ioremap_nocache(EXTINT1,4);                         //映射为虚拟地址
intmsk=ioremap nocache(INTMSK,4);
s3c2410_gpio_cfgpin(S3C2410_GPF7, S3C2410_GPF7_EINT7);      //设置外部中断 7
writel(readl(extint1)|0x40,extint1);                        //上升沿
writel(readl(intmsk)&0xfff1,intmsk);
iounmap(intmsk);                                            //取消映射
iounmap(extint1);
```

### 5．编译内核

进入源码目录，输入 make menuconfig 进入内核的配置选项菜单，按以下顺序选择选项：Device Drivers-->Network device surport-->Ethernet(10 or 100Mbit) -->DM9000 support ，如图 9.10 所示。

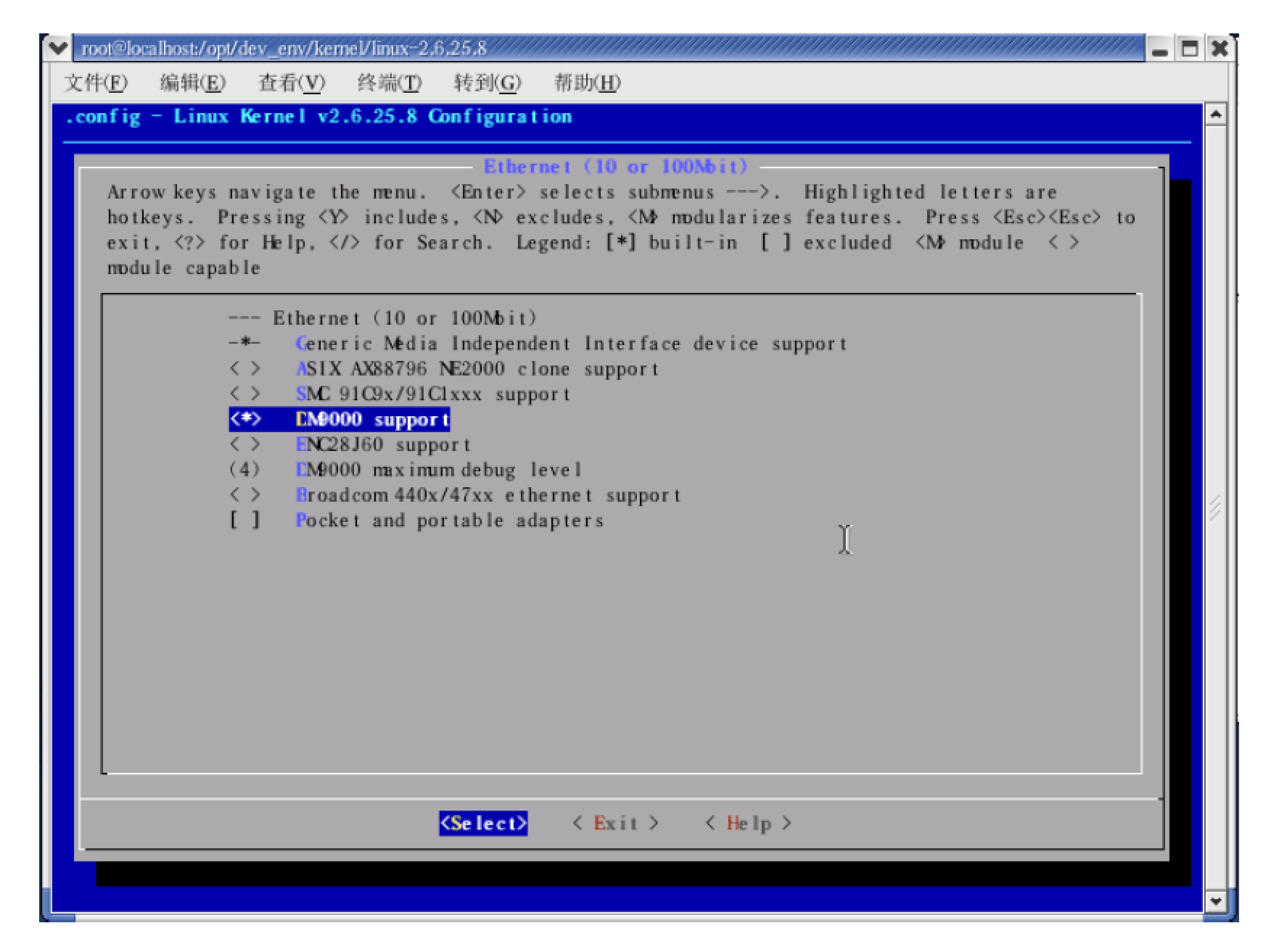



图 9.10　内核配置选项图

## 9.6　小　　结

本章主要在分析 DM9000 硬件操作和内核中自带的 DM9000 驱动源程序的基础上，讲述了 DM9000 网卡驱动的移植。做网格驱动程序移植主要是要了解内核中网络设备驱动程序的体系结构，所以，本章的 9.2 节是后面移植工作的基础，读者要认真掌握，特别是对网络数据结构的主要成员要掌握其代表的具体含义。

# 第 10 章　音频设备驱动程序移植

音频驱动程序广泛地应用在嵌入式产品中，目前 PDA、手机都有音频和视频播放功能。随着终端产品逐渐融入工作和生活，带有音频功能的嵌入式产品将更具吸引力。本章将介绍音频设备接口分类、Linux 音频设备驱动、音频设备应用程序编写、音频设备驱动移植、最后介绍音频播放程序 madplay 的移植。

## 10.1　音频设备接口

Linux 支持的音频设备接口包括 PCM、IIS 和 AC97。这 3 种接口分别应用在不同的场合，IIS 接口多应用在 MP3 随身听、CD 等产品中；PCM 接口多应用在移动电话中；AC97 接口多用在 PDA 中。

### 10.1.1　PCM（脉冲编码调制）接口

PCM 接口针对不同的数字音频子系统，用于数字转换的接口，它是最简单的音频接口，该接口由时钟脉冲（BCLK）、帧同步信号（FS）及接收数据（DR）和发送数据（DX）组成。在 FS 信号的上升沿，数据传输从 MSB（Most Significant Bit）字开始，FS 频率等于采样频率。在 FS 信号之后开始传输数据字，按顺序进行传输单个的数据位，每时钟周期传输 1 个数据字。发送 MSB 时，首先将信号的等级降到最低，以避免在不同终端的接口使用不同的数据方案时造成 MSB 的丢失。

PCM 接口的特点是：容易实现，原则上能够支持任何数据方案和任何采样频率，但需要每个音频通道获得一个独立的数据队列。

### 10.1.2　IIS（Inter-IC Sound）接口

IIS 接口在 20 世纪 80 年代首先被飞利浦用于消费音频，并在一个称为 LRCLK（Left/Right CLOCK）的信号机制中经过多路转换，将两路音频信号合成单一的数据队列。当 LRCLK 信号为高时，左声道数据被传输；LRCLK 信号为低时，右声道数据被传输。

IIS 接口的特点：与 PCM 相比，IIS 接口更适用于立体声系统。对于多通道系统，在同样的 BCLK 和 LRCLK 条件下，也可以并行执行几个数据队列。

### 10.1.3　AC97（Audio Codec 1997）接口

AC97（Audio Codec 1997）是以 Intel 为首的 5 个 PC 厂商（Intel、Creative Labs、NS、

Analog Device 与 Yamaha）共同提出的规格标准。与 PCM 和 IIS 不同，AC97 除了是一种数据格式还具有控制功能，用于音频编码的内部架构规范。AC97 采用 AC-Link 与外部的编解码器相连，AC-Link 接口包括位时钟（BITCLK）、同步信号校正（SYNC）和从编码到处理器及从处理器中解码(SDATDIN 与 SDATAOUT)的数据队列。AC97 数据帧以 SYNC 脉冲开始，包括 12 个 20 位时间段（时间段为标准中定义的不同的目的服务）及 16 位 tag 段，共计 256 个数据序列。例如，时间段 1 和 2 用于访问编码的控制寄存器，而时间段 3 和 4 分别负载左、右两个音频通道。tag 段表示其他段中哪一个包含有效数据。把帧分成时间段使传输控制信号和音频数据仅通过 4 根线到达 9 个音频通道或转换成其他数据流成为可能。

AC97 特点：与控制接口分离的 IIS 方案相比，AC97 明显减少了整体管脚数。它是一种数据格式还具有控制功能。

## 10.1.4　Linux 音频设备驱动框架

针对音频设备，Linux 内核附有两类音频设备驱动框架：OSS（Open Sound System）和 ALSA（Advanced Linux Sound Architecture）。前者包含 dev/dsp 和 dev/mixer 字符设备接口，在用户空间的编程中，使用文件操作；后者以 card 和组件（pcm、mixer 等）为主线，在用户空间的编程中不使用文件接口而使用 alsalib。在音频设备驱动中，几乎都用到 DMA，DMA 的缓冲区会被分割成多段，每次 DMA 操作进行其中的一个段。OSS 驱动框架的阻塞读写操作具有流控能力，在用户空间不需要进行流量方面的定时工作，但是它需要及时的写（播放）和读（录音），避免缓冲区 underflow 或 overflow。

在内核配置时，选择 Device Drivers |Sound card support 命令进入 Sound card support 配置窗口。该窗口中包含 OSS 与 ALSA 驱动架构配置选择，如图 10.1 所示。

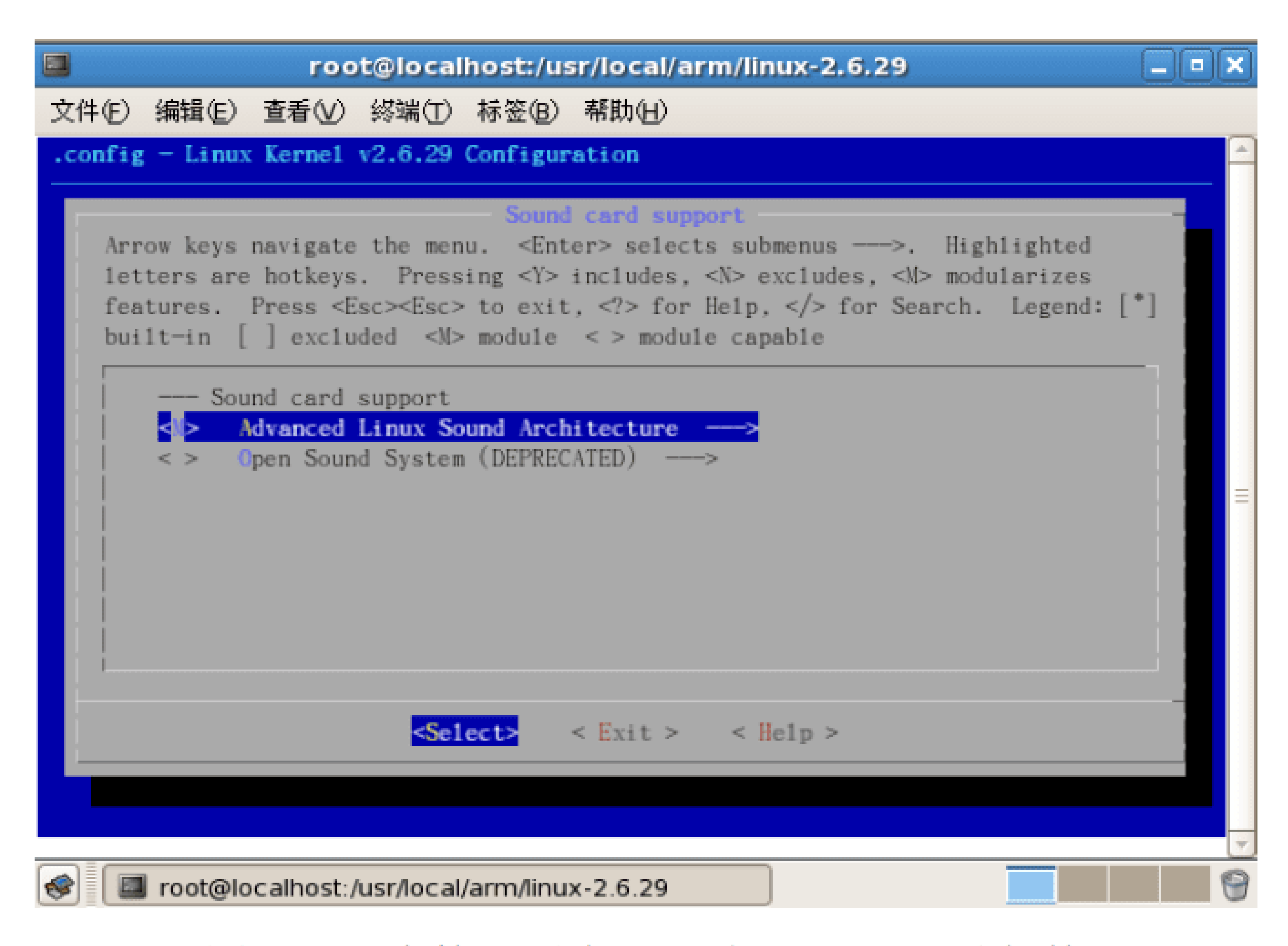



图 10.1　内核配置中 OSS 与 ALSA 驱动架构

**注意：** overflow 是指应用程序读取数据的速率低于声卡的采样速率，多余的数据被丢弃；underflow 是指应用程序读取数据的速率高于声卡的采样速率，那么在新的数据到来之前，声卡将会阻塞应用程序的请求。

前面提到了 Linux 两类音频驱动框架：OSS 和 ALSA，下面将分别介绍这两类驱动架构，并且通过代码进行分析其如何实现录音和播放等功能。

# 10.2　Linux 音频设备驱动——OSS 驱动框架

OSS（Open Sound System）是 UNIX 平台上一个统一的音频接口。为了在不同的平台之间移植代码，OSS 定义了一套 API，在一个平台上的代码移植到另一个平台上时，不需要重新修改代码，重新编译就可以使用原程序。

## 10.2.1　OSS 驱动架构硬件

数字音频（有时也称 CODEC、PCM、DSP、ADC/DAC 设备）接口，用来实现录音（将模拟信号转变为数字信号）和播放声音（将数字信号转变为模拟信号）的功能。它的主要参数有：采样频率（电话为 8K，DVD 为 96K）、channel 数目（单声道，立体声）、采样分辨率（8-bit，16-bit），对应的设备文件为/dev/dsp。OSS 驱动支持的硬件接口有以下几种。

- mixer（混频器）接口：用来控制多个输入、输出的音量，也控制输入（microphone，line-in，CD）之间的切换，对应的设备文件为/dev /mixer。
- synthesizer（合成器）接口：通过一些预先定义好的波形来合成声音，有时用在游戏中声音效果的产生。
- MIDI（Musical Intrument Data Interface）接口：MIDI 接口是为了连接舞台上的 synthesizer、键盘、道具、灯光控制器的一种串行接口。

## 10.2.2　OSS 驱动架构代码

当 vwsnd 驱动模块被加载时函数 init_vwsnd()被调用，音频驱动被初始化，驱动初始化过程会查找硬件配置和匹配相应的驱动程序。函数 init_vwsnd()的主要内容如下：

```
static int __init init_vwsnd(void)
{
    //驱动的探针函数
    if (!probe_vwsnd(&the_hw_config))
        return -ENODEV;
    //驱动附着函数
    err = attach_vwsnd(&the_hw_config);
}
```

在函数 attach_vwsnd()中，调用了对数字音频设备的注册函数 register_sound_dsp()和对混频器接口的注册函数 register_sound_mixer ()。函数的主要内容如下：

```
static int __init attach_vwsnd(struct address_info *hw_config)
{
```

```
    vwsnd_dev_t *devc = NULL;
    int err = -ENOMEM;
    //为 devc 分配空间
    devc = kmalloc(sizeof (vwsnd_dev_t), GFP_KERNEL);
    if (devc == NULL)
        goto fail0;
    //下面是给结构体指针 devc 各个字段赋值
    err = li_create(&devc->lith, hw_config->io_base);
    if (err)
        goto fail1;

    init_waitqueue_head(&devc->open_wait);

    devc->rport.hwbuf_size = HWBUF_SIZE;
    devc->rport.hwbuf_vaddr = __get_free_pages(GFP_KERNEL, HWBUF_ORDER);
    if (!devc->rport.hwbuf_vaddr)
        goto fail2;
    devc->rport.hwbuf = (void *) devc->rport.hwbuf_vaddr;
    devc->rport.hwbuf_paddr = virt_to_phys(devc->rport.hwbuf);

    /*设置输入侧的 DMA 基地址，保持它到驱动被卸载前*/
    li_writel(&devc->lith, LI_COMM1_BASE,
          devc->rport.hwbuf_paddr >> 8 | 1 << (37 - 8));

    devc->wport.hwbuf_size = HWBUF_SIZE;
    devc->wport.hwbuf_vaddr = __get_free_pages(GFP_KERNEL, HWBUF_ORDER);
    if (!devc->wport.hwbuf_vaddr)
        goto fail3;
    devc->wport.hwbuf = (void *) devc->wport.hwbuf_vaddr;
    devc->wport.hwbuf_paddr = virt_to_phys(devc->wport.hwbuf);
    DBGP("wport hwbuf = 0x%p\n", devc->wport.hwbuf);

    DBGDO(shut_up++);
    err = ad1843_init(&devc->lith);
    DBGDO(shut_up--);
    if (err)
        goto fail4;

    /*安装中断处理程序*/
    err = request_irq(hw_config->irq, vwsnd_audio_intr, 0, "vwsnd", devc);
    if (err)
        goto fail5;

    /*注册 dsp 设备驱动程序*/
    devc->audio_minor = register_sound_dsp(&vwsnd_audio_fops, -1);
    if ((err = devc->audio_minor) < 0) {
        DBGDO(printk(KERN_WARNING
                "attach vwsnd: register_sound_dsp error %d\n",
                err));
        goto fail6;
    }
    /*注册 mixer 设备驱动程序*/
    devc->mixer_minor = register_sound_mixer(&vwsnd_mixer_fops,
                    devc->audio_minor >> 4);
    if ((err = devc->mixer_minor) < 0) {
        DBGDO(printk(KERN_WARNING
                "attach vwsnd: register_sound_mixer error %d\n",
                err));
        goto fail7;
    }
```

```
    /*Squirrel away device indices for unload routine.*/
    hw_config->slots[0] = devc->audio_minor;
    /*对设备执行初始化工作*/
    mutex_init(&devc->open_mutex);
    mutex_init(&devc->io_mutex);
    mutex_init(&devc->mix_mutex);
    devc->open_mode = 0;
    spin_lock_init(&devc->rport.lock);
    init_waitqueue_head(&devc->rport.queue);
    devc->rport.swstate = SW_OFF;
    devc->rport.hwstate = HW_STOPPED;
    devc->rport.flags = 0;
    devc->rport.swbuf = NULL;
    spin_lock_init(&devc->wport.lock);
    init_waitqueue_head(&devc->wport.queue);
    devc->wport.swstate = SW_OFF;
    devc->wport.hwstate = HW_STOPPED;
    devc->wport.flags = 0;
    devc->wport.swbuf = NULL;

    /*注册成功后，将新设备连接到本地设备。*/
    devc->next_dev = vwsnd_dev_list;
    vwsnd_dev_list = devc;
    return devc->audio_minor;
    /*如果注册、分配空间、安装等过程失败，则释放这些过程分配的资源。*/
 fail7:
    unregister_sound_dsp(devc->audio_minor);
 fail6:
    free_irq(hw_config->irq, devc);
 fail5:
 fail4:
    free_pages(devc->wport.hwbuf_vaddr, HWBUF_ORDER);
 fail3:
    free_pages(devc->rport.hwbuf_vaddr, HWBUF_ORDER);
 fail2:
    li_destroy(&devc->lith);
 fail1:
    kfree(devc);
 fail0:
    return err;
}
```

OSS 的驱动程序在 sound/oss 目录下，核心代码文件为 soundcard.c。该文件中定义了打开设备文件，读设备文件（录音），写设备文件（播放），关闭设备文件等功能。

## 10.2.3　OSS 初始化函数 oss_init()

在向内核注册该模块的时候调用该函数对 OSS 进行初始化工作，也就是当使用 insmod soundcore.ko 加载驱动模块时调用该函数。

```
static int __init oss_init(void)
{
    int             err;
    int i, j;

#ifdef CONFIG_PCI
```

```
    if(dmabug)
        isa dma bridge buggy = dmabug;
#endif
    /*函数 create_special_devices()最终调用函数 register_sound_special_dev-
    ice()注册声音结点，根据第二个参数来指定声音结点的类型。这里指定注册了两个设备
    sequencer 和 sequencer2。*/
    err = create special devices();
    if (err) {
        printk(KERN_ERR "sound: driver already loaded/included in kernel
        \n");
        return err;
    }

    /*Protecting the innocent*/
    sound_dmap_flag = (dmabuf > 0 ? 1 : 0);
    /*创建设备列表中的设备，并且注册它到系统文件中。*/
    for (i = 0; i < ARRAY SIZE(dev list); i++) {
        device_create(sound_class, NULL,
                 MKDEV(SOUND_MAJOR, dev_list[i].minor), NULL,
                 "%s", dev_list[i].name);

        if (!dev list[i].num)
            continue;
        /*如果设备列表的某项数目多于 1 个则创建剩下的设备，并且注册它们到系统文件中*/
        for (j = 1; j < *dev_list[i].num; j++)
            device create(sound class, NULL,
                     MKDEV(SOUND_MAJOR,
                       dev_list[i].minor + (j*0x10)),
                     NULL, "%s%d", dev list[i].name, j);
    }

    if (sound_nblocks >= 1024)
        printk(KERN_ERR "Sound warning: Deallocation table was too small.
        \n");

    return 0;
}
```

## 10.2.4　OSS 释放函数 oss_cleanup()

从内核中注销模块 OSS 的时候调用该函数对 OSS 进行清理工作，也就是当使用 rmmod soundcore.ko 卸载驱动模块时调用该函数。

```
static void __exit oss_cleanup(void)
{
    int i, j;
    /*删除函数 device_create()创建的设备。*/
    for (i = 0; i < ARRAY_SIZE(dev_list); i++) {
        device_destroy(sound_class, MKDEV(SOUND_MAJOR, dev_list[i].min-
        or));
        if (!dev list[i].num)
            continue;
        for (j = 1; j < *dev_list[i].num; j++)
            device_destroy(sound_class, MKDEV(SOUND_MAJOR, dev_list[i]
            .minor + (j*0x10)));
    }
```

```
    /*在初始化函数 oss_init()中调用函数 create_special_devices （），在函数
    create_special_devices ()中分别注册了设备 sequencer 和 sequencer2，在注销函
    数中对应注销这两个设备的驱动程序。*/
    unregister_sound_special(1);
    unregister_sound_special(8);
    /*停止定时器*/
    sound_stop_timer();
    /*音序器卸载，释放资源*/
    sequencer_unload();
    /*释放 DMA 资源*/
    for (i = 0; i < MAX_DMA_CHANNELS; i++)
        if (dma_alloc_map[i] != DMA_MAP_UNAVAIL) {
           printk(KERN_ERR "Sound: Hmm, DMA%d was left allocated - fixed\n",
            i);
            sound_free_dma(i);
        }
    /*释放静态分配的内存资源*/
    for (i = 0; i < sound_nblocks; i++)
        vfree(sound_mem_blocks[i]);

}
```

## 10.2.5　打开设备文件函数 sound_open()

当驱动模块注册到系统中后，调用系统的 open()函数就会调用函数 sound_open()实现打开设备文件。

```
static int sound_open(struct inode *inode, struct file *file)
{
    int dev = iminor(inode);
    int retval;

    DEB(printk("sound_open(dev=%d)\n", dev));
    if ((dev >= SND_NDEVS) || (dev < 0)) {
        printk(KERN_ERR "Invalid minor device %d\n", dev);
        return -ENXIO;
    }
    switch (dev & 0x0f) {
    /*音频设备接口为控制设备即混频器*/
    case SND_DEV_CTL:
        dev >>= 4;
        if (dev >= 0 && dev < MAX_MIXER_DEV && mixer_devs[dev] == NULL) {
            request_module("mixer%d", dev);
        }
        if (dev && (dev >= num_mixers || mixer_devs[dev] == NULL))
            return -ENXIO;

        if (!try_module_get(mixer_devs[dev]->owner))
            return -ENXIO;
        break;
    /*打开的设备接口为音序器*/
    case SND_DEV_SEQ:
    case SND_DEV_SEQ2:
        if ((retval = sequencer_open(dev, file)) < 0)
            return retval;
        break;
    /*打开的设备为MIDI*/
```

```
    case SND_DEV_MIDIN:
        if ((retval = MIDIbuf_open(dev, file)) < 0)
            return retval;
        break;
    /*打开的设备接口为 AUDIO 或者 DSP*/
    case SND_DEV_DSP:
    case SND_DEV_DSP16:
    case SND_DEV_AUDIO:
        if ((retval = audio_open(dev, file)) < 0)
            return retval;
        break;

    default:
        printk(KERN_ERR "Invalid minor device %d\n", dev);
        return -ENXIO;
    }

    return 0;
}
```

### 10.2.6　录音函数 sound_read()

在录音时，调用函数 sound_read()实现读设备文件。

```
static ssize_t sound_read(struct file *file, char __user *buf, size_t count,
loff_t *ppos)
{
    int dev = iminor(file->f_path.dentry->d_inode);
    int ret = -EINVAL;

    /*锁住内核*/
    lock_kernel();

    DEB(printk("sound_read(dev=%d, count=%d)\n", dev, count));
    /*设备接口类型为 AUDIO 或者 DSP 时，调用函数 audio_read ()读取设备文件数据到 buf
    中*/
    switch (dev & 0x0f) {
    case SND_DEV_DSP:
    case SND_DEV_DSP16:
    case SND_DEV_AUDIO:
        ret = audio_read(dev, file, buf, count);
        break;
    /*设备接口类型为音序器时，调用函数 sequencer_read ()读取设备文件数据到 buf 中*/
    case SND_DEV_SEQ:
    case SND_DEV_SEQ2:
        ret = sequencer_read(dev, file, buf, count);
        break;
    /*设备接口类型为 MIDI 时，调用函数 MIDIbuf_read ()读取设备文件数据到 buf*/
    case SND_DEV_MIDIN:
        ret = MIDIbuf_read(dev, file, buf, count);
    }
    /*完成读取后解锁内核*/
    unlock_kernel();
    return ret;
}
```

## 10.2.7　播放函数 sound_write()

在播放音频时，调用函数 sound_write()向设备文件执行写操作。

```
static ssize_t sound_write(struct file *file, const char __user *buf, size_t
count, loff_t *ppos)
{
    int dev = iminor(file->f_path.dentry->d_inode);
    int ret = -EINVAL;
    /*锁住内核*/
    lock_kernel();
    DEB(printk("sound_write(dev=%d, count=%d)\n", dev, count));
    switch (dev & 0x0f) {
    /*设备接口类型为音序器时，调用函数 sequencer_ write ()将 buf 中数据写到到设备文
    件*/
    case SND_DEV_SEQ:
    case SND_DEV_SEQ2:
        ret =  sequencer_write(dev, file, buf, count);
        break;
    /*设备接口类型为 AUDIO 或者 DSP 时，调用函数 audio_write ()将 buf 中数据写到到设
    备文件*/
    case SND_DEV_DSP:
    case SND_DEV_DSP16:
    case SND_DEV_AUDIO:
        ret = audio_write(dev, file, buf, count);
        break;
    /*设备接口类型为 MIDI 时，调用函数 MIDIbuf_write ()将 buf 中数据写到到设备文件*/
    case SND_DEV_MIDIN:
        ret =  MIDIbuf_write(dev, file, buf, count);
        break;
    }
    unlock_kernel();
    return ret;
}
```

## 10.2.8　控制函数 sound_ioctl()

函数 sound_ioctl()控制功能包括：音量控制、低音控制、高音控制、FM 合成器控制、录音音量、播放音量、输入增益、输出增益等控制。

```
static int sound_ioctl(struct inode *inode, struct file *file,
               unsigned int cmd, unsigned long arg)
{
    int len = 0, dtype;
    int dev = iminor(inode);
    void __user *p = (void __user *)arg;

    if ( SIOC_DIR(cmd) !=  SIOC_NONE &&  SIOC_DIR(cmd) != 0) {
        /*
         * Have to validate the address given by the process.
        */
        len =  SIOC_SIZE(cmd);
        if (len < 1 || len > 65536 || !p)
```

```
            return -EFAULT;
        if ( SIOC_DIR(cmd) & _SIOC_WRITE)
            if (!access_ok(VERIFY_READ, p, len))
                return -EFAULT;
        if (_SIOC_DIR(cmd) & _SIOC_READ)
            if (!access_ok(VERIFY_WRITE, p, len))
                return -EFAULT;
    }
    DEB(printk("sound_ioctl(dev=%d, cmd=0x%x, arg=0x%x)\n", dev, cmd,
    arg));
    if (cmd == OSS_GETVERSION)
        return __put_user(SOUND_VERSION, (int __user *)p);
    /*如果命令的类型为Mixer，调用sound_mixer_ioctl ()*/
    if (_IOC_TYPE(cmd) == 'M' && num_mixers > 0 &&   /*Mixer ioctl*/
        (dev & 0x0f) != SND_DEV_CTL) {
        dtype = dev & 0x0f;
        switch (dtype) {
        case SND_DEV_DSP:
        case SND_DEV_DSP16:
        case SND_DEV_AUDIO:
            return sound_mixer_ioctl(audio_devs[dev >> 4]->mixer_dev, cmd,
            p);
        default:
            return sound_mixer_ioctl(dev >> 4, cmd, p);
        }
    }
    switch (dev & 0x0f) {
    /*设备接口类型为AUDIO 或者DSP 时，调用函数sound_mixer_ioctl ()*/
    case SND_DEV_CTL:
        if (cmd == SOUND_MIXER_GETLEVELS)
            return get_mixer_levels(p);
        if (cmd == SOUND_MIXER_SETLEVELS)
            return set_mixer_levels(p);
        return sound_mixer_ioctl(dev >> 4, cmd, p);
    /*设备接口类型为音序器时，调用函数sequencer_ioctl ()*/
    case SND_DEV_SEQ:
    case SND_DEV_SEQ2:
        return sequencer_ioctl(dev, file, cmd, p);
    /*设备接口类型为AUDIO 或者DSP 时，调用函数audio_ioctl ()*/
    case SND_DEV_DSP:
    case SND_DEV_DSP16:
    case SND_DEV_AUDIO:
        return audio_ioctl(dev, file, cmd, p);
        break;
    /*设备接口类型为MIDI 时，调用函数MIDIbuf_ioctl ()*/
    case SND_DEV_MIDIN:
        return MIDIbuf_ioctl(dev, file, cmd, p);
        break;

    }
    return -EINVAL;
}
```

## 10.3　Linux 音频设备驱动——ALSA 驱动框架

ALSA 包含了许多的声卡驱动程序，提供 libasound 的 API 库。libasound 提供了方便

的高级编程接口，应用程序中使用 libasound 而不是内核中的 ALSA 接口。同时，libasound 提供一个设备逻辑命名功能，因此开发者不必知道类似设备文件这样的底层接口。相反，OSS 驱动是在内核系统调用级上编程，要求应用程序开发者提供设备文件名并且利用 ioctl 来实现相应的功能。为了兼容 OSS，ALSA 框架使用内核模块来模拟 OSS，在 OSS 基础上开发的应用程序不需要任何改动就可以在 ALSA 上运行。另外，libaoss 库也能够模拟 OSS，而它不需要内核模块。ALSA 包含插件功能，使用插件可以扩展新的声卡驱动，包括完全使用软件实现的虚拟声卡。ALSA 提供一系列基于命令行的工具集 alsa-utils，比如 amixer（混音器命令行控制），aplay（音频文件命令行播放器），arecord（命令行音频文件录制）等工具。

ALSA 提供给用户的接口主要有：

- 信息接口（Information Interface, /proc/asound）。
- 控制接口（Control Interface, /dev/snd/controlCX）提供管理声卡注册和请求可用设备的通用功能。
- PCM 接口（PCM Interface, /dev/snd/pcmCXDX）管理数字音频回放（playback）和录音（capture）的接口。
- Raw MIDI 接口（Raw MIDI Interface, /dev/snd/midiCXDX）支持 MIDI（Musical Instrument Digital Interface），标准的电子乐器。这些 API 提供对声卡上 MIDI 总线的访问。
- 定时器接口（Timer Interface，/dev/snd/timer）为同步音频事件提供对声卡上时间处理硬件的访问。
- 音序器接口（Sequencer Interface, /dev/snd/seq）。
- 混音器接口（Mixer Interface）。

## 10.3.1　card 和组件

每个声卡必须创建一个 card 实例，下面通过驱动代码介绍 ALSA 声卡驱动是如何管理 card 和组件的。

### 1. 创建card函数snd_card_new()

函数 snd_card_new()用于创建和初始化 snd_card 结构体。idx 是卡的索引号，xid 是标识字符串，module 是顶层模块，extra_siz 额外分配数据的大小。

```
struct snd_card *snd_card_new(int idx, const char *xid, struct module *module,
int extra size)
{
    struct snd_card *card;
    int err, idx2;

    if (extra size < 0)
        extra_size = 0;
    /*分配空间同时初始化所分配的空间*/
    card = kzalloc(sizeof(*card) + extra_size, GFP_KERNEL);
    if (card == NULL)
        return NULL;
```

```
    if (xid) {
        if (!snd_info_check_reserved_words(xid))
            goto   error;
        strlcpy(card->id, xid, sizeof(card->id));
    }
    /*锁住互斥量*/
    mutex_lock(&snd_card_mutex);
    if (idx < 0) {
        for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
            /*idx == -1 == 0xffff means: take any free slot*/
            if (~snd_cards_lock & idx & 1<<idx2) {
                if (module_slot_match(module, idx2)) {
                    idx = idx2;
                    break;
                }
            }
    }
    if (idx < 0) {
        for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
            /*idx == -1 == 0xffff means: take any free slot*/
            if (~snd_cards_lock & idx & 1<<idx2) {
                if (!slots[idx2] || !*slots[idx2]) {
                    idx = idx2;
                    break;
                }
            }
    }
    snd_cards_lock |= 1 << idx;     /*lock it*/
    if (idx >= snd_ecards_limit)
        snd_ecards_limit = idx + 1; /*increase the limit*/
    /*解锁互斥量*/
    mutex_unlock(&snd_card_mutex);
    /*对 card 各个字段进行初始化和赋值*/
    card ->number = idx;
    card->module = module;
    INIT_LIST_HEAD(&card->devices);
    init_rwsem(&card->controls_rwsem);
    rwlock_init(&card->ctl_files_rwlock);
    INIT_LIST_HEAD(&card->controls);
    INIT_LIST_HEAD(&card->ctl_files);
    spin_lock_init(&card->files_lock);
    init_waitqueue_head(&card->shutdown_sleep);
    return card;                                /*省略了错误处理部分*/
}
```

### 2．创建组件函数snd_ device _new()

在 card 被创建后，创建设备（组件）关联到该 card。参数 card 为 snd_card_new()创建的 card，参数 type 为设备类型，包括 SNDRV_DEV_TOPLEVEL、SNDRV_DEV_CONTROL、SNDRV_DEV_LOWLEVEL_PRE、SNDRV_DEV_LOWLEVEL_NORMAL、SNDRV_DEV_PCM、SNDRV_DEV_RAWMIDI、SNDRV_DEV_TIMER、SNDRV_DEV_SEQUENCER、SNDRV_DEV_CODEC 等（具体的定义在 include/sound/core.h 中），device_data 为设备数据指针，ops 为函数集的指针。

```
int snd_device_new(struct snd_card *card, snd_device_type_t type,
          void *device_data, struct snd_device_ops *ops)
```

```
{
    struct snd_device *dev;

    if (snd_BUG_ON(!card || !device_data || !ops))
        return -ENXIO;
    /*给设备分配空间并初始化*/
    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
    if (dev == NULL) {
        snd_printk(KERN_ERR "Cannot allocate device\n");
        return -ENOMEM;
    }
    /*给设备的各个字段赋值*/
    dev->card = card;
    dev->type = type;
    dev->state = SNDRV_DEV_BUILD;
    dev->device_data = device_data;
    dev->ops = ops;
    /*将设备列表插入到 card 的设备字段后*/
    list_add(&dev->list, &card->devices);  /*add to the head of list*/
    return 0;
}
```

### 3．释放组件函数snd_device_free ()

函数 snd_device_free ()释放与 card 关联的设备，并且对应设备的数据为：device_data。

```
int snd_device_free(struct snd_card *card, void *device_data)
{
    struct snd_device *dev;

    if (snd_BUG_ON(!card || !device_data))
        return -ENXIO;
    /*根据 device_data 找到要释放的设备*/
    list_for_each_entry(dev, &card->devices, list) {
        if (dev->device_data != device_data)
            continue;
        /*该设备从链表中断开*/
        list_del(&dev->list);
        /*判断断开是否成功*/
        if (dev->state == SNDRV_DEV_REGISTERED && dev->ops->dev_disconnect)
            if (dev->ops->dev_disconnect(dev))
                snd_printk(KERN_ERR "device disconnect failure\n");
        /*判断释放是否成功*/
        if (dev->ops->dev_free) {
            if (dev->ops->dev_free(dev))
                snd_printk(KERN_ERR "device free failure\n");
        }
        /*释放设备占用的内存*/
        kfree(dev);
        return 0;
    }
    snd_printd("device free %p (from %pF), not found\n", device_data,
          __builtin_return_address(0));
    return -ENXIO;
}
```

### 4．释放card函数snd_card_free()

该函数释放 snd_card_new()创建和初始化的 snd_card 结构体。

```
int snd_card_free(struct snd_card *card)
{
    /*释放操作，并通知连接在 card 上的所有设备*/
    int ret = snd_card_disconnect(card);
    if (ret)
        return ret;

    /*所有的设备准备好释放操作*/
    wait_event(card->shutdown_sleep, card->files == NULL);
    /*释放 card*/
    snd_card_do_free(card);
    return 0;
}
```

### 5．断开所有的ALSA API函数snd_card_disconnect ()

在系统中 file_operations 中对应系统的操作 llseek 与 card 的操作 snd_disconnect_llseek，系统的 read 与 snd_disconnect_read，系统的 write 与 snd_disconnect_write 等，函数 snd_card_disconnect ()的作用是将这些对应关系断开，并通知该信息到所有连接在 card 上的设备。

```
int snd_card_disconnect(struct snd_card *card)
{
    /*第一阶段：使 ALSA API 操作失效*/
    mutex_lock(&snd_card_mutex);
    snd_cards[card->number] = NULL;
    snd_cards_lock &= ~(1 << card->number);
    mutex_unlock(&snd_card_mutex);
    /*第二阶段：用专门的哑操作代替 file->f_op*/
    spin_lock(&card->files_lock);
    mfile = card->files;
    while (mfile) {
        file = mfile->file;
        /*it's critical part, use endless loop*/
        /*we have no room to fail*/
        mfile->disconnected_f_op = mfile->file->f_op;
        spin_lock(&shutdown_lock);
        list_add(&mfile->shutdown_list, &shutdown_files);
        spin_unlock(&shutdown_lock);
        mfile->file->f_op = &snd_shutdown_f_ops;
        fops_get(mfile->file->f_op);
        mfile = mfile->next;
    }
    spin_unlock(&card->files_lock);
    /*第三阶段：通知所有的连接设备断开信息*/
    snd_device_disconnect_all(card);
    snd_info_card_disconnect(card);
    return 0;
}
```

## 10.3.2　PCM 设备

每个声卡设备最多有 4 个 PCM 实例，而每个 PCM 实例对应一个设备文件。每个 PCM 实例由 PCM 播放流和 PCM 录音流组成，每个 PCM 流由一个或多个 PCM 子流组成。

### 1．函数snd_pcm_new()用来创建PCM实例

函数 snd_pcm_new()中的各个参数的作用分别为：参数 card 是与 PCM 对应的声卡，参数 id 为标识字符串，参数 device 是 PCM 设备索引用来表示第几个 PCM 设备，参数 playback_count 为播放的子流数，参数 capture_count 为录音的子流数，参数 rpcm 用来返回构造的 PCM 实例。

```
int snd_pcm_new(struct snd_card *card, char *id, int device,
        int playback_count, int capture_count, struct snd_pcm ** rpcm)
{
    struct snd_pcm *pcm;
    int err;
    static struct snd_device_ops ops = {
        .dev_free = snd_pcm_dev_free,
        .dev_register = snd_pcm_dev_register,
        .dev_disconnect = snd_pcm_dev_disconnect,
    };
    /*为构造的 PCM 实例分配空间并进行初始化*/
    pcm = kzalloc(sizeof(*pcm), GFP_KERNEL);
    if (pcm == NULL) {
        snd_printk(KERN_ERR "Cannot allocate PCM\n");
        return -ENOMEM;
    }
    /*指定 PCM 的声卡与 PCM 设备索引*/
    pcm->card = card;
    pcm->device = device;
    if (id)
        strlcpy(pcm->id, id, sizeof(pcm->id));
    /*构造 PCM 播放流*/
    if ((err = snd_pcm_new_stream(pcm, SNDRV_PCM_STREAM_PLAYBACK,
    playback_count)) < 0) {
        snd_pcm_free(pcm);
        return err;
    }
    /*构造 PCM 录音流*/
    if ((err = snd_pcm_new_stream(pcm, SNDRV_PCM_STREAM_CAPTURE,
    capture_count)) < 0) {
        snd_pcm_free(pcm);
        return err;
    }
    mutex_init(&pcm->open_mutex);
    init_waitqueue_head(&pcm->open_wait);
    /*创建设备类型为 SNDRV_DEV_PCM 的设备，并指定该设备的声卡为 card*/
    if ((err = snd_device_new(card, SNDRV_DEV_PCM, pcm, &ops)) < 0) {
        snd_pcm_free(pcm);
        return err;
    }
    if (rpcm)
        *rpcm = pcm;
```

```
    return 0;
}
```

### 2．函数snd_pcm_ free ()释放PCM实例

函数 snd_pcm_ free ()用来释放 snd_pcm_new()创建的 PCM 实例。释放包括释放播放流和录音流。

```
static int snd_pcm_free(struct snd_pcm *pcm)
{
    struct snd_pcm_notify *notify;

    list_for_each_entry(notify, &snd_pcm_notify_list, list) {
        notify->n_unregister(pcm);
    }
    if (pcm->private free)
        pcm->private free(pcm);
    /*释放分配的缓冲区*/
    snd pcm lib preallocate free for all(pcm);
    /*释放播放流*/
    snd_pcm_free_stream(&pcm->streams[SNDRV_PCM_STREAM_PLAYBACK]);
    /*释放录音流*/
    snd_pcm_free_stream(&pcm->streams[SNDRV_PCM_STREAM_CAPTURE]);
    kfree(pcm);
    return 0;
}
```

### 3．设置PCM操作

函数 snd_pcm_set_ops()设置 PCM 打开子流、关闭子流等操作。

```
void snd_pcm_set_ops(struct snd_pcm *pcm, int direction, struct snd_pcm_ops
*ops)
{
    struct snd_pcm_str *stream = &pcm->streams[direction];
    struct snd_pcm_substream *substream;
    /*设置子流操作*/
    for (substream = stream->substream; substream != NULL; substream =
    substream->next)
        substream->ops = ops;
}
```

snd_pcm_ops 结构体定义如下：

```
struct snd pcm ops {
    int (*open)(struct snd_pcm_substream *substream);      //打开子流
    int (*close)(struct snd_pcm_substream *substream);     //关闭子流
    int (*ioctl)(struct snd_pcm_substream * substream,
            unsigned int cmd, void *arg);                  //I/O 控制
    int (*hw_params)(struct snd_pcm_substream *substream,
            struct snd_pcm_hw_params *params);             //硬件参数
    int (*hw_free)(struct snd_pcm_substream *substream);   //释放资源
    int (*prepare)(struct snd_pcm_substream *substream);   //准备
    int (*trigger)(struct snd_pcm_substream *substream, int cmd);
                                        //在 PCM 开始、停止、暂停时调用
    snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream);
```

```
                                        //当前缓冲区的硬件位置
    int  (*copy)(struct snd_pcm_substream *substream, int channel,
           snd_pcm_uframes_t pos,
           void __user *buf, snd_pcm_uframes_t count);     //复制缓冲区
    int  (*silence)(struct snd_pcm_substream *substream, int channel,
             snd_pcm_uframes_t pos, snd_pcm_uframes_t count); //静音
    struct page *(*page)(struct snd_pcm_substream *substream,
               unsigned long offset);                      //cache 操作
    int (*mmap)(struct snd_pcm_substream *substream, struct vm_area_struct
    *vma);                                                 //内存映射
    int  (*ack)(struct snd_pcm_substream *substream);      //应答
};
```

### 4. PCM运行时结构体

打开 PCM 子流后，分配 PCM 运行时实例给该子流。运行时实例的结构体定义如下：

```
struct snd_pcm_runtime {
    /*子流的状态*/
    struct snd_pcm_substream *trigger_master;
    struct timespec trigger_tstamp;          //触发时间戳
    int overrange;
    snd_pcm_uframes_t avail_max;
    snd_pcm_uframes_t hw_ptr_base;           //缓冲区复位时位置
    snd_pcm_uframes_t hw_ptr_interrupt;      //缓冲区中断时位置

    /*子流硬件参数*/
    snd_pcm_access_t access;                 //访问模式
    snd_pcm_format_t format;                 //SNDRV_PCM_FORMAT_*
    snd_pcm_subformat_t subformat;           //子格式
    unsigned int rate;                       //速率单位 Hz
    unsigned int channels;                   //通道
    snd_pcm_uframes_t period_size;           //周期大小
    unsigned int periods;                    //周期数
    snd_pcm_uframes_t buffer_size;           //缓冲区大小
    snd_pcm_uframes_t min_align;             //对齐的最小块大小
    size_t byte_align;                       //字节对齐
    unsigned int frame_bits;                 //每帧的比特数
    unsigned int sample_bits;                //采样的比特数（编码所使用的比特数）
    unsigned int info;
    unsigned int rate_num;
    unsigned int rate_den;

    /*软件参数*/
    int tstamp_mode;                         //更新内存映射时间戳
    unsigned int period_step;
    snd_pcm_uframes_t start_threshold;
    snd_pcm_uframes_t stop_threshold;
    snd_pcm_uframes_t silence_threshold;     //Silence 填充阈值
    snd_pcm_uframes_t silence_size;          //Silence 填充大小
    snd_pcm_uframes_t boundary;              //封装指针的指针
    snd_pcm_uframes_t silence_start;         //silence 区域开始指针
    snd_pcm_uframes_t silence_filled;        //silence 被填充的大小
    union snd_pcm_sync_id sync;              //硬件同步 ID
```

```
    /*内存映射*/
    struct snd_pcm_mmap_status *status;     //内存映射状态
    struct snd_pcm_mmap_control *control;   //内存映射控制
    /*锁和调度*/
    wait queue head t sleep;
    struct fasync_struct *fasync;
    /*私有段*/
    void *private_data;
    void (*private_free)(struct snd_pcm_runtime *runtime);
    /*硬件描述*/
    struct snd pcm hardware hw;
    struct snd_pcm_hw_constraints hw_constraints;
    /*中断回调函数*/
    void (*transfer_ack_begin)(struct snd_pcm_substream *substream);
    void (*transfer_ack_end)(struct snd_pcm_substream *substream);
    /*定时器*/
    unsigned int timer_resolution;          //定时器精度
    int tstamp_type;                        //时间戳类型
    /*DMA 缓冲区信息*/
    unsigned char *dma_area;                //DMA 区域*/
    dma_addr_t dma_addr;                    //总线物理地址*/
    size_t dma_bytes;                       //DMA 区域大小*/
    struct snd_dma_buffer *dma_buffer_p;    //分配的缓冲区
#if defined(CONFIG SND PCM OSS) || defined(CONFIG SND PCM OSS MODULE)
    /*OSS 运行时结构体*/
    struct snd pcm oss runtime oss;
#endif
};
```

### 10.3.3　控制接口

控制接口（control）的主要作用是混音器（mixer），所有 mixer 元素都是基于 control API 实现的。下面简单介绍 control 的几个重要函数。

#### 1．snd_ctl_open()

函数 snd_ctl_open()用于打开设备文件操作。

```
static int snd_ctl_open(struct inode *inode, struct file *file)
{
    unsigned long flags;
    struct snd_card *card;
    struct snd_ctl_file *ctl;
    /*从注册的设备获得用户数据*/
    card = snd_lookup_minor_data(iminor(inode), SNDRV_DEVICE_TYPE_
    CONTROL);
    /*增加文件到 card 的文件链表*/
    snd card file add(card, file);
    /*为 ctl 分配空间并初始化*/
    ctl = kzalloc(sizeof(*ctl), GFP_KERNEL);
    /*初始化 ctl 事件链表*/
    INIT_LIST_HEAD(&ctl->events);
```

```
    /*对 ctl 的各个字段进行初始化或者赋值*/
    init_waitqueue_head(&ctl->change_sleep);
    spin_lock_init(&ctl->read_lock);
    ctl->card = card;
    ctl->prefer_pcm_subdevice = -1;
    ctl->prefer_rawmidi_subdevice = -1;
    ctl->pid = current->pid;
    file->private_data = ctl;
    //对链表操作前进行锁保护
    write_lock_irqsave(&card->ctl_files_rwlock, flags);
    list_add_tail(&ctl->list, &card->ctl_files);
    //完成操作进行解锁
    write_unlock_irqrestore(&card->ctl_files_rwlock, flags);
    return 0;
}
```

### 2．创建控制实例snd_ctl_new()

函数 snd_ctl_new()创建控制实例，参数 control 为控制模板，参数 access 为默认的控制访问。

```
static struct snd_kcontrol *snd_ctl_new(struct snd_kcontrol *control,
                        unsigned int access)
{
    struct snd_kcontrol *kctl;
    unsigned int idx;

    if (snd_BUG_ON(!control || !control->count))
        return NULL;
    //分配空间并初始化
    kctl = kzalloc(sizeof(*kctl) + sizeof(struct snd_kcontrol_volatile) *
    control->count, GFP_KERNEL);
    if (kctl == NULL) {
        snd_printk(KERN_ERR "Cannot allocate control instance\n");
        return NULL;
    }
    //复制控制模板
    *kctl = *control;
    for (idx = 0; idx < kctl->count; idx++)
        kctl->vd[idx].access = access;
    return kctl;
}
```

### 3．移植控制函数snd_ctl_remove()

函数 snd_ctl_remove()从 card 移除 control，并且释放它占用的空间。

```
int snd_ctl_remove(struct snd_card *card, struct snd_kcontrol *kcontrol)
{
    struct snd_ctl_elem_id id;
    unsigned int idx;

    if (snd_BUG_ON(!card || !kcontrol))
        return -EINVAL;
    /*移除控制链表*/
    list_del(&kcontrol->list);
    card->controls_count -= kcontrol->count;
```

```
    id = kcontrol->id;
    for (idx = 0; idx < kcontrol->count; idx++, id.index++, id.numid++)
        snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_REMOVE, &id);
    /*释放空间*/
    snd_ctl_free_one(kcontrol);
    return 0;
}
```

## 10.3.4　AC97 API 音频接口

ALSA AC97 已经有各种接口，驱动开发人员通过编写少量的控制函数就可以开发自己的驱动。

### 1．函数snd_ac97_bus()创建AC97 bus总线组件

函数 snd_ac97_bus()创建 AC97 bus 总线组件及 AC97 的操作。

```
int snd_ac97_bus(struct snd_card *card, int num, struct snd_ac97_bus_ops
*ops,
         void *private_data, struct snd_ac97_bus **rbus)
{
    int err;
    struct snd_ac97_bus *bus;
    static struct snd_device_ops dev_ops = {
        .dev_free = snd_ac97_bus_dev_free,
    };
    /*分配空间并初始化*/
    bus = kzalloc(sizeof(*bus), GFP_KERNEL);
    /*bus 各字段赋值*/
    bus->card = card;
    bus->num = num;
    bus->ops = ops;
    bus->private_data = private_data;
    bus->clock = 48000;
    spin_lock_init(&bus->bus_lock);
    /*总线初始化*/
    snd_ac97_bus_proc_init(bus);
    /*构造 BUS 类型的设备，并指定声卡为 card*/
    if ((err = snd_device_new(card, SNDRV_DEV_BUS, bus, &dev_ops)) < 0) {
        snd_ac97_bus_free(bus);
        return err;
    }
    if (rbus)
        *rbus = bus;
    return 0;
}
```

### 2．创建Codec97 组件函数snd_ac97_mixer()

在创建完 AC97 bus 后，调用该函数创建依附于该 bus 的 Codec97 组件。下面列出该函数的主要部分代码：

```
int  snd_ac97_mixer(struct  snd_ac97_bus  *bus,  struct  snd_ac97_template
*template, struct snd_ac97 **rac97)
```

```
{
    int err;
    struct snd_ac97 *ac97;
    struct snd_card *card;
    char name[64];
    unsigned long end_time;
    unsigned int reg;
    const struct ac97_codec_id *pid;
    static struct snd_device_ops ops = {
        .dev_free = snd_ac97_dev_free,
        .dev_register = snd_ac97_dev_register,
        .dev_disconnect =  snd_ac97_dev_disconnect,
    };

    if (rac97)
        *rac97 = NULL;
    /*设置 bus 的 card 字段为该声卡*/
    card = bus->card;
    /*为创建的组件 AC97 分配空间并初始化*/
    ac97 = kzalloc(sizeof(*AC97), GFP_KERNEL);
    /*初始化或者设置所创建的组件 AC97 的各个字段*/
    ac97->private_data = template->private_data;
    ac97->private_free = template->private_free;
    ac97->bus = bus;
    ac97->pci = template->pci;
    ac97->num = template->num;
    ac97->addr = template->addr;
    ac97->scaps = template->scaps;
    ac97->res_table = template->res_table;
    bus->codec[ac97->num] = ac97;
    /*信号量初始化*/
    mutex_init(&ac97->reg_mutex);
    mutex_init(&ac97->page_mutex);
    /*bus 复位*/
    if (bus->ops->reset) {
        bus->ops->reset(ac97);
        goto __access_ok;
    }
    /*读取产品的 ID，AC97_VENDOR_ID1 放置在高两字节，AC97_VENDOR_ID2 放置在低两字
    节*/
    ac97->id = snd_ac97_read(ac97, AC97_VENDOR_ID1) << 16;
    ac97->id |= snd_ac97_read(ac97, AC97_VENDOR_ID2);
    if (ac97->id && ac97->id != (unsigned int)-1) {
        pid = look_for_codec_id(snd_ac97_codec_ids, ac97->id);
        if (pid && (pid->flags & AC97_DEFAULT_POWER_OFF))
            goto __access_ok;
    }
    /*恢复为默认*/
    if (!(ac97->scaps & AC97_SCAP_SKIP_AUDIO))
        snd_ac97_write(ac97, AC97_RESET, 0);
    if (!(ac97->scaps & AC97_SCAP_SKIP_MODEM))
        snd_ac97_write(ac97, AC97_EXTENDED_MID, 0);
    if (bus->ops->wait)
        bus->ops->wait(ac97);
    else {
        /*延迟 50 微秒*/
        udelay(50);
        /*等待直到寄存器复位至可以访问*/
        if (ac97->scaps & AC97_SCAP_SKIP_AUDIO)
```

```
            err = ac97_reset_wait(ac97, msecs_to_jiffies(500), 1);
        else {
            err = ac97_reset_wait(ac97, msecs_to_jiffies(500), 0);
            if (err < 0)
                err = ac97_reset_wait(ac97, msecs_to_jiffies(500), 1);
        }
        if (err < 0) {
            snd_printk(KERN_WARNING "AC'97 %d does not respond - RESET\n",
            ac97->num);
            /*proceed anyway - it's often non-critical*/
        }
    }
       access_ok:
    /*读取产品 ID 信息*/
    ac97->id = snd_ac97_read(ac97, AC97_VENDOR_ID1) << 16;
    ac97->id |= snd_ac97_read(ac97, AC97_VENDOR_ID2);
    if (! (ac97->scaps & AC97_SCAP_DETECT_BY_VENDOR) &&
        (ac97->id == 0x00000000 || ac97->id == 0xffffffff)) {
        snd_printk(KERN_ERR "AC'97 %d access is not valid [0x%x], removing
        mixer.\n", ac97->num, ac97->id);
        snd_ac97_free(ac97);                    //若为无效组件则释放该组件
        return -EIO;
    }
    pid = look_for_codec_id(snd_ac97_codec_ids, ac97->id);
    if (pid)
        ac97->flags |= pid->flags;

    /*为接口 AC'97 测试该组件*/
    if (!(ac97->scaps & AC97_SCAP_SKIP_AUDIO) && !(ac97->scaps & AC97_
    SCAP_AUDIO)) {
        /*测试是否可以向 AC97 的 AC97_REC_GAIN 寄存器写值 0x8a06*/
        snd_ac97_write_cache(AC97, AC97_REC_GAIN, 0x8a06);
        /*测试读取 AC97 的 AC97_REC_GAIN 寄存器的值与 0x7fff 做与运算的结果是否为
        0x0a06*/
        if (((err = snd_ac97_read(ac97, AC97_REC_GAIN)) & 0x7fff) == 0x0a06)
            ac97->scaps |= AC97_SCAP_AUDIO;
    }
    /*读取 AC97_RESET 和 AC97_EXTENDED_ID 的值分别赋给 caps 和 ext_id 字段*/
    if (ac97->scaps & AC97_SCAP_AUDIO) {
        ac97->caps = snd_ac97_read(ac97, AC97_RESET);
        ac97->ext_id = snd_ac97_read(ac97, AC97_EXTENDED_ID);
        if (ac97->ext_id == 0xffff) /*无效组件*/
            ac97->ext_id = 0;
    }
    /*测试 MC'97，读取 AC97_EXTENDED_MID 寄存器并测试读取的有效性*/
    if  (!(ac97->scaps  &  AC97_SCAP_SKIP_MODEM)  &&  !(ac97->scaps  &
    AC97_SCAP_MODEM)) {
        ac97->ext_mid = snd_ac97_read(ac97, AC97_EXTENDED_MID);
        if (ac97->ext_mid == 0xffff)    /*无效组件*/
            ac97->ext_mid = 0;
        if (ac97->ext_mid & 1)
            ac97->scaps |= AC97_SCAP_MODEM;
    }
    /*检查 AC97 的 audio 与 modem*/
    if (!ac97_is_audio(ac97) && !ac97_is_modem(ac97)) {
        if (!(ac97->scaps & (AC97_SCAP_SKIP_AUDIO|AC97_SCAP_SKIP_MODEM)))
            snd_printk(KERN_ERR "AC'97 %d access error (not audio or modem
            codec)\n", ac97->num);
        snd_ac97_free(ac97);//如果 audio 与 modem 不正确，则释放创建的 AC97 组件
```

```
        return -EACCES;
    }
    /*/*设置 mixername 名字*/
    if (ac97_is_audio(ac97)) {
        /*获得组件 ID 字符串*/
        char comp[16];
        if (card->mixername[0] == '\0') {
            strcpy(card->mixername, name);
        } else {
            if (strlen(card->mixername) + 1 + strlen(name) + 1 <= sizeof
            (card->mixername)) {
                strcat(card->mixername, ",");
                strcat(card->mixername, name);
            }
        }
        sprintf(comp, "AC97a:%08x", ac97->id);
        /*添加组件 ID 字符串 comp 到所给的链表*/
        if ((err = snd_component_add(card, comp)) < 0) {
            snd_ac97_free(ac97);                    //如添加失败,则释放该组件 AC97
            return err;
        }
        /*构造控制器 mixer,该构造过程比较复杂,构造内容比较多,包括 master controls、
        center controls 、LFE controls 、surround controls 、headphone
        controls 、mono controls (单声道控制器) 、tone controls (音质控制器)、
        PC Speaker controls (PC 扬声器)、Phone controls 、MIC controls (麦
        克风)、Line controls 、CD controls 、PCM controls 等。创建失败则释放
        组件 AC97。*/
        if (snd_ac97_mixer_build(ac97) < 0) {
            snd_ac97_free(ac97);
            return -ENOMEM;
        }
    }
    if (ac97_is_modem(ac97)) {
        /*获得组件 ID 字符串*/
        char comp[16];
        if (card->mixername[0] == '\0') {
            strcpy(card->mixername, name);
        } else {
            if (strlen(card->mixername) + 1 + strlen(name) + 1 <= sizeof
            (card->mixername)) {
                strcat(card->mixername, ",");
                strcat(card->mixername, name);
            }
        }
        sprintf(comp, "AC97m:%08x", ac97->id);
        /*添加组件 ID 字符串 comp 到所给的链表*/
        if ((err = snd_component_add(card, comp)) < 0) {
            snd_ac97_free(ac97);
            return err;
        }
        /*构造 modem ,如构造失败则释放 AC97*/
        if (snd_ac97_modem_build(card, ac97) < 0) {
            snd_ac97_free(ac97);
            return -ENOMEM;
        }
    }
    /*更新 power 寄存器*/
    if (ac97_is_audio(ac97))
```

```
        update_power_regs(ac97);
    snd ac97 proc init(ac97);
    /*构造设备类型为 SNDRV_DEV_CODEC 的设备，并设置设备的 card 为该声卡，如果失败则
    释放 AC97*/
    if ((err = snd device new(card, SNDRV DEV CODEC, ac97, &ops)) < 0) {
        snd_ac97_free(ac97);
        return err;
    }
    *rac97 = ac97;
    return 0;
}
```

### 3. 释放bus函数snd_ac97_bus_dev_free()

函数 snd_ac97_bus_dev_free()释放函数 snd_ac97_bus()，创建 AC97 bus 总线。

```
static int snd_ac97_bus_dev_free(struct snd_device *device)
{
    struct snd_ac97_bus *bus = device->device_data;
    return snd ac97 bus free(bus);
}
```

# 10.4　音频设备应用程序编写

10.3 节中介绍了 OSS 驱动和 ALSA 驱动，本节主要介绍如何编写这两种驱动程序的应用程序。对于 OSS 驱动，给出了 DSP 接口编程和 MIXER 接口编程。

## 10.4.1　DSP 接口编程

OSS 驱动框架提供了音频编程的 3 种设备，分别是/dev/dsp、/dev/dspW 和/dev/audio。用户可以直接使用命令播放和录音，命令 cat /dev/dsp >test 可用来录音，录音的结果放在 test 文件中；命令 cat test>/dev/dsp 播放声音文件 test。OSS 应用程序主要包括打开设备、录音、播放、设置参数等部分，下面分别介绍。

### 1. 包含相关头文件

在使用 OSS 驱动编程时，应该包含相应的头文件，需要包含的头文件如下：

```
#include <ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/soundcard.h>
```

### 2. 打开设备文件

在对设备进行操作前，打开设备。打开的设备包括/dev/dsp、dev/dspW 和/dev/audio，如下面的设备名 DEVICE_NAME 可以设为“dev/dsp”。打开设备的模式包括 O_RDONLY、_WRONLY 和 O_RDWR，分别表示只读、只写和读写。

```
#define BUF_SIZE 4096
int fd;
unsigned char buf[BUF_SIZE];
if ((fd = open(DEVICE_NAME, O_RDONLY, 0)) == -1) {
  /*如果打开设备失败则退出程序*/
  perror(DEVICE_NAME);
  exit(1);
}
```

### 3．录音

read()函数中参数 count 为录音数据的字节个数（建议为 2 的指数，如 512），但不能大于 buf 的大小。从读字节的个数可以精确地算时间，例如采样频率为 8kHz，采样精度为 8bit 的立体声的速率为 8000×1×2=16KBytes/second，这是计算停止录音时间的唯一方法。

```
int len;
if ((len = read(fd, buf, count)) == -1) {
  perror("audio read");
  exit(1);
}
```

### 4．播放

播放实际上和录音很类似，相应的 buf 中为音频数据，sizeof (buf)为数据的长度。注意，用户始终要读/写一个完整的采样。例如一个采样精度 16bit 的立体声模式下，每个采样有 4 个字节，所以应用程序每次必须读/写 4 的倍数个字节。

```
if (write (fd, buf, sizeof (buf)) != sizeof (buf))
{
  perror ("Audio write");
  exit (-1);
}
```

**注意：** 由于 OSS 是一个跨平台的音频接口，所以用户在编程的时候，要考虑到可移植性的问题，其中一个重要的方面是读/写时的字节顺序。

### 5．设置参数

参数设置包括设置缓冲区大小、设置采用格式、设置通道数目和设置采样速率等。

（1）设置缓冲区大小

Linux 内核中的声卡驱动程序专门维护了一个缓冲区，其大小影响到放音和录音时的效果，使用 ioctl 系统调用可以对它的大小进行调整。调节驱动程序中缓冲区大小的操作不是必须的，如果没有特殊的要求，缓冲区大小通常采用默认设置。并且缓冲区大小的设置通常应紧跟在打开设备文件之后，因为对声卡的其他操作有可能会导致驱动程序无法再修改其缓冲区的大小。设置声卡驱动程序中内核缓冲区的大小方法如下：

```
int size = 0xnnnnssss;
int result = ioctl(handle, SNDCTL_DSP_SETFRAGMENT, &size);
if (result == -1) {
    perror("ioctl buffer size");
    return -1;
}
```

在设置缓冲区大小时，参数 size 由两部分组成，其低 16 位标明缓冲区的尺寸，相应的计算公式为 size = $2^{ssss}$，即若参数 size 低 16 位的值为 16，那么相应的缓冲区的大小会被设置为 $2^{16}$=65536 字节。参数 size 的高 16 位则用来标明分片（fragment）的最大序号，它的取值范围从 2 到 0x7FFF，其中 0x7FFF 表示没有任何限制。

（2）设置采样格式

在音频编程时需要考虑采样格式和采样频率，在头文件 soundcard.h 中定义声卡支持的所有采样格式，而通过系统调用 ioctl 则可以很方便地更改当前所使用的采样格式。下面的代码示范对声卡的采样格式进行设置的方法：

```
int format;
format = AFMT_S16_NE; /*Native 16 bits*/
if (ioctl(fd, SNDCTL_DSP_SETFMT, &format) == -1) {
  /*fatal error*/
  perror("SNDCTL DSP SETFMT");
  exit(1);
}
if (format != AFMT_S16_NE) {
  /*本设备不支持选择的采样格式*/
}
/*在设置采样格式之前，可以先测试设备能够支持哪些采样格式，方法如下：*/
int mask;
if (ioctl(fd, SNDCTL_DSP_GETFMTS, &mask) == -1) {
  /*Handle fatal error ...*/
}
if (mask & AFMT MPEG) {
  /*本设备支持 MPEG 采样格式*/
}
```

（3）设置通道数目

```
int channels = 2;              /*1=mono, 2=stereo*/
if (ioctl(fd, SNDCTL_DSP_CHANNELS, &channels) == -1) {
  /*Fatal error*/
  perror("SNDCTL_DSP_CHANNELS");
  exit(1);
}
if (channels != 2){
  /*本设备不支持立体声模式*/
}
```

（4）设置采样速率

声卡采样频率的设置比较简单，调用 ioctl 时设置第 2 个参数为 SNDCTL_DSP_SPEED，同时在第 3 个参数中指定采样频率的数值即可。对于大多数声卡来说，其支持的采样频率范围一般为 5kHz 到 44.1kHz 或者 48kHz，但并不意味着该范围内的所有频率都会被硬件支持。在 Linux 下进行音频编程时最常用到的几种采样频率是 11025Hz、16000Hz、22050Hz、32000Hz 和 44100Hz。下面的代码示范对声卡的采样频率进行设置：

```
int rate = 11025;
if (ioctl(fd, SNDCTL_DSP_SPEED, & rate)==-1) {
  /*Fatal error*/
  perror("SNDCTL_DSP_SPEED");
  exit(Error code);
}
```

## 10.4.2　MIXER 接口编程

声卡上的混音器由多个混音通道组成，它们可以通过驱动程序提供的设备文件/dev/mixer 进行编程。对混音器的操作是通过系统调用 ioctl 来完成，所有的混音器控制命令都以 SOUND_MIXER 或者 MIXER 开头，表 10.1 中列出了常用的混音器控制命令。

**表 10.1　常用的混音器控制命令**

| 命　　令 | 作　　用 |
|---|---|
| SOUND_MIXER_VOLUME | 调节主音量 |
| SOUND_MIXER_BASS | 低音控制 |
| SOUND_MIXER_TREBLE | 高音控制 |
| SOUND_MIXER_SYNTH | FM 合成器 |
| SOUND_MIXER_PCM | 主 D/A 转换器 |
| SOUND_MIXER_SPEAKER | PC 喇叭 |
| SOUND_MIXER_LINE | 音频线输入 |
| SOUND_MIXER_MIC | 麦克风输入 |
| SOUND_MIXER_CD | CD 输入 |
| SOUND_MIXER_IMIX | 回放音量 |
| SOUND_MIXER_ALTPCM | 从 D/A |
| SOUND_MIXER_RECLEV | 录音音量 |
| SOUND_MIXER_IGAIN | 输入增益 |
| SOUND_MIXER_OGAIN | 输出增益 |
| SOUND_MIXER_LINE1 | 声卡的第 1 输入 |
| SOUND_MIXER_LINE2 | 声卡的第 2 输入 |
| SOUND_MIXER_LINE3 | 声卡的第 3 输入 |

### 1．声卡的输入增益和输出增益调节

混音器的一个主要作用是对声卡的输入增益和输出增益进行调节，目前大部分声卡采用的是 8 位或者 16 位的增益控制器，程序员不需要关心这些，因为声卡驱动程序将它们变换成百分比的形式，即无论是输入增益还是输出增益，其取值范围都是从 0～100。在使用混音器编程时，可以通过使用宏 SOUND_MIXER_READ 来读取混音通道的增益大小，下面是获取麦克风的输入增益的方法：

```
ioctl(fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol);
```

如果设备只有一个混音通道的单声道，那么返回的增益大小保存在低位字节中。如果设备支持多个混音通道的双声道，那么返回的增益大小包括左、右声道值两个部分，左声道的音量保存在低位字节，而右声道的音量则保存高位字节。下面是提取左、右声道的增益大小示例代码。

```
int left, right;
left = vol & 0xff;
right = (vol & 0xff00) >> 8;
```

通过 SOUND_MIXER_WRITE 宏来设置混音通道的增益大小，与获取增益值时基本相

同，下面是设置麦克风的输入增益：

```
vol = (right << 8) + left;
ioctl(fd, SOUND_MIXER_WRITE(SOUND_MIXER_MIC), &vol);
```

### 2. 查询混音器的信息

声卡驱动程序提供了多个系统调用 ioctl 来获得混音器的信息，返回一个整型的位掩码（bitmask），掩码的每一位分别代表一个特定的混音通道，如果相应的位为 1，则说明与之对应的混音通道是可用的。下面是通过 SOUND_MIXER_READ_RECMASK 返回的位掩码检查 CD 输入是否为一个有效的混音通道。

```
ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
if (devmask & SOUND_MIXER_CD)
  printf("The CD input is supported");
```

检查 CD 输入是否为有效的录音源代码。

```
ioctl(fd, SOUND_MIXER_READ_RECMASK, &recmask);
if (recmask & SOUND_MIXER_CD)
  printf("The CD input can be a recording source");
```

通过使用宏 SOUND_MIXER_READ_RECSRC 可以查询当前声卡正在使用的录音源。

```
if (ioctl(mixer_fd, SOUND_MIXER_READ_RECSRC, &mask) == -1)
{
  perror("/dev/mixer");
}
printf("%x\n",mask);
```

通过使用宏 SOUND_MIXER_WRITE_RECSRC 对当前声卡使用的录音源进行设置，下面是将 CD 输入作为声卡的录音源使用的代码。

```
devmask = SOUND_MIXER_CD;
ioctl(fd, SOUND_MIXER_WRITE_DEVMASK, &devmask);
```

可以通过 SOUND_MIXER_READ_STEREODEVS 查询混音通道是否对立体声提供支持。

## 10.4.3　ALSA 应用程序编程

对于 ALSA 应用程序编程，用户不用使用文件接口，可以使用 alsa-lib。下面给出一个播放 PCM 流的小程序及其编译和测试方法。代码文件名为 talsa.cpp，内容如下：

```
#include <iostream>
#include <alsa/asoundlib.h>
using namespace std;

int main(int argc, char *argv[])
{
      long                       loops;
      int                        rc;
      int                        size;
      snd_pcm_t*                 handle;              //PCM 设备句柄
```

```
snd_pcm_hw_params_t*       params;          //硬件信息和 PCM 流配置
unsigned int               val= 22050; //采样频率
int                        dir;
snd_pcm_uframes_t          frames= 16; //采样精度
char*                      buffer;
/*打开 PCM，最后一个参数为 0 表示标准配置*/
if ( (rc = snd_pcm_open(&handle, "default", SND_PCM_STREAM_PLAYBACK,
0)) < 0)
{
      cerr << "unable to open pcm device: " << snd_strerror(rc) << endl;
      exit(1);
}
/*分配 snd_pcm_hw_params_t 结构体*/
snd_pcm_hw_params_alloca(&params);
/*初始化 hw_params*/
snd_pcm_hw_params_any(handle, params);
/*初始化访问权限*/
snd_pcm_hw_params_set_access(handle, params, SND_PCM_ACCESS_RW_
INTERLEAVED);
/*初始化采样格式*/
snd_pcm_hw_params_set_format(handle, params, SND_PCM_FORMAT_S16_
LE);
/*设置通道数*/
snd_pcm_hw_params_set_channels(handle, params, 2);
/*设置采样率*/
snd_pcm_hw_params_set_rate_near(handle, params, &val, &dir);
/*设置周期大小*/
snd_pcm_hw_params_set_period_size_near(handle, params, &frames,
&dir);
/*设置 hw_params*/
if ( (rc = snd_pcm_hw_params(handle, params)) < 0)
{
      cerr << "unable to set hw paramseters: " << snd_strerror(rc) <<
      endl;
      exit(1);
}

snd_pcm_hw_params_get_period_size(params, &frames, &dir);
size = frames * 4;
buffer = new char[size];
/*获得周期*/
snd_pcm_hw_params_get_period_time(params, &val, &dir);
/*3 秒钟的数据*/
loops = 3000000 / val;

while (loops > 0) {
      loops--;
      if ( (rc = read(0, buffer, size)) == 0)
      {
            cerr << "end of file on input" << std::endl;
            break;
      }
      else if (rc != size)
            cerr << "short read: read " << rc << " bytes" << endl;

      if ( (rc = snd_pcm_writei(handle, buffer, frames)) == -EPIPE)
      {
            cerr << "underrun occurred" << endl;
```

```
                    snd_pcm_prepare(handle);
              }
              else if (rc < 0)
                    cerr << "error from writei: " << snd_strerror(rc) << endl;
              else if (rc != (int)frames)
                    cerr << "short write, write " << rc << " frames" << endl;
        }
       /*把所有挂起没有传输完的声音样本传输完全*/
        snd_pcm_drain(handle);
        /*关闭该音频流*/
        snd_pcm_close(handle);
        /*释放之前动态分配的缓冲区*/
        free(buffer);

        return 0;
}
```

编译上面的 ALSA 应用程序采用下面的命令：

```
#g++ -o talsa talsa.cpp -lasound
```

运行应用程序测试，下面的命令可以产生随机的白噪声。

```
./talsa </dev/urandom
```

也可以使用 Windows 的录音工具，打开“开始”|“所有程序”|“附件”|“娱乐”|“录音机”，录制一段 wav 格式文件，默认的音频格式是 PCM 22.050 kHz，16 位，立体声。使用上面编译的 ALSA 应用程序播放录制的音频文件的方法为：

```
# ./talsa < rev.wav
```

# 10.5　音频设备驱动移植

目前笔者手中的 mini2440 使用的是 UDA1341 芯片，本节就通过移植 UDA1341 讲解移植音频设备驱动的过程。

## 10.5.1　添加 UDA1341 结构体

如果买了友善之臂 mini2440 的用户，那么在附带的源代码中已经完成了这部分工作。如果读者的内核是从网上下载的，请下载 Linux-2.6.29 版本以上的内核，因为笔者测试的时候采用的是 Linux-2.6.29 版本，交叉编译器采用的是 arm-linux-gcc-4.3.2。修改的文件为 arch/arm/mach-s3c2440 目录下的 mach-mini2440.c，修改的内容如下：

（1）在 mach-mini2440.c 中包括头文件。

```
#include <sound/s3c24xx_uda134x.h>
```

（2）在 mach-mini2440.c 中添加 UDA1341 设备结构。

```
static struct s3c24xx_uda134x_platform_data s3c24xx_uda134x_data = {
    .l3_clk = S3C2410_GPB4,
    .l3_data = S3C2410_GPB3,
```

```
    .l3_mode = S3C2410_GPB2,
    .model = UDA134X_UDA1341,
};
static struct platform_device s3c24xx_uda134x = {
    .name = "s3c24xx_uda134x",
    .dev = {
        .platform_data    = &s3c24xx_uda134x_data,
    }
};
```

（3）在下面的结构体中，添加注册 UDA1341 设备平台到内核。

```
static struct platform_device *mini2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_rtc,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
    &s3c_device_dm9k,
    &net_device_cs8900,
    &s3c24xx_uda134x,
};
```

## 10.5.2　修改录音通道

Mini2440 使用录音通道是 VIN2，应该修改 sound/soc/codecs 目录下的 uda134x.c 文件，在函数 uda134x_startup 中修改录音通道为 VIN2。

```
static int uda134x_startup(struct snd_pcm_substream *substream,
    struct snd_soc_dai *dai)
{
    struct snd_soc_pcm_runtime *rtd = substream->private_data;
    struct snd_soc_device *socdev = rtd->socdev;
    struct snd_soc_codec *codec = socdev->codec;
    struct uda134x_priv *uda134x = codec->private_data;
    struct snd_pcm_runtime *master_runtime;

    if (uda134x->master_substream) {
        master_runtime = uda134x->master_substream->runtime;

        pr_debug("%s constraining to %d bits at %d\n", __func__,
             master_runtime->sample_bits,
             master_runtime->rate);

        snd_pcm_hw_constraint_minmax(substream->runtime,
                     SNDRV_PCM_HW_PARAM_RATE,
                     master_runtime->rate,
                     master_runtime->rate);

        snd_pcm_hw_constraint_minmax(substream->runtime,
                     SNDRV_PCM_HW_PARAM_SAMPLE_BITS,
                     master_runtime->sample_bits,
                     master_runtime->sample_bits);
```

```
        uda134x->slave substream = substream;
    } else
        uda134x->master substream = substream;

    //修改录音通道为 VIN2
    uda134x_write(codec, 2, 2|(5U<<2));
    return 0;
}
```

## 10.5.3　内核中添加 UDA1341 驱动支持

上面对内核代码进行了修改，回到内核代码一级目录下，使用命令 make menuconfig 对内核以窗口的方式进行配置内核。进入内核配置界面后选择 Device Drivers| Sound card support|Advanced Linux Sound Architecture 命令进入 ALSA 驱动配置界面，选择 ALSA for SoC audio support，同时选上对 Mixer API 和 PCM API 的支持，如图 10.2 所示。

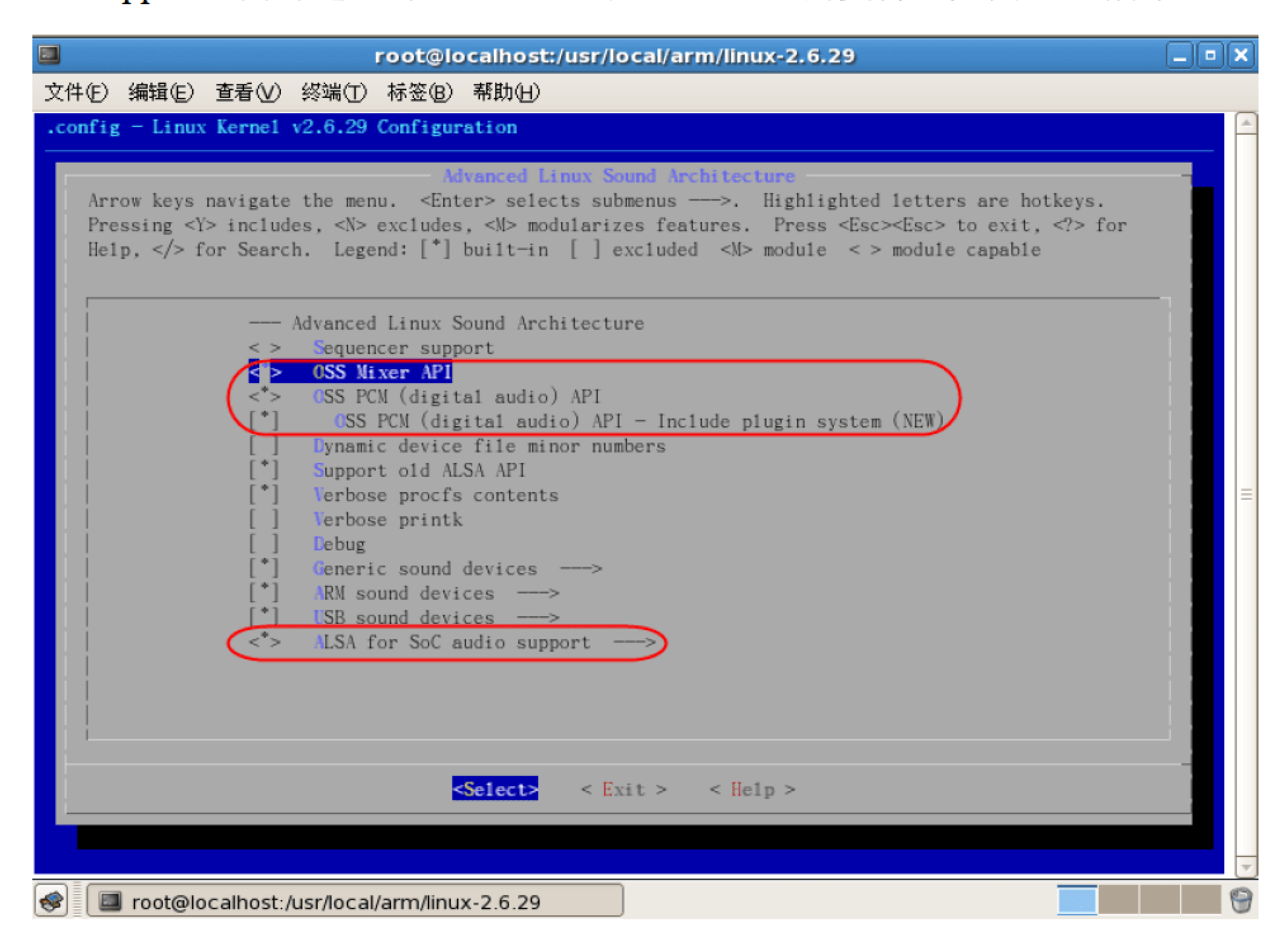



图 10.2　内核配置 ALSA for SoC audio support

**注意：** 如果不添加 OSS PCM (digital audio) API 支持，在开发板上播放音乐时会出现无法找到设备的错误“audio: /dev/dsp: No such file or directory”。

进入 ALSA for SoC audio support 的配置界面后，选择 SoC I2S Audio support UDA134X wired to a S3C24XX，如图 10.3 所示。

保存配置后，使用 make clean 清除以前编译的临时文件，再使用 make zImage 编译内核。

```
# make clean
# make zImage
```

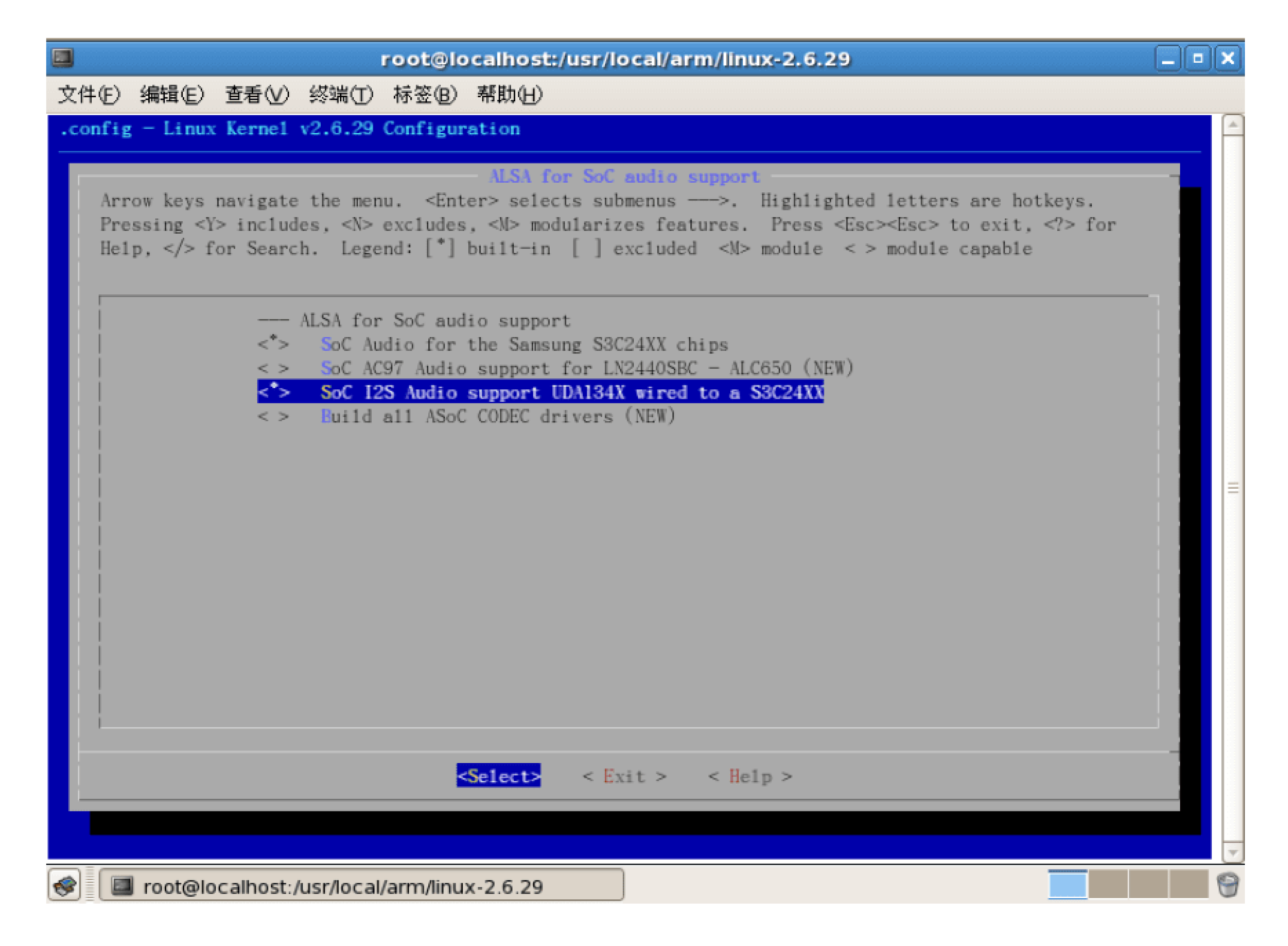



图 10.3　配置 UDA134X 芯片的驱动

## 10.5.4　移植新内核并进行测试

将新生成的内核 zImage 使用命令 load flash kernel u 下载到开发板上。因为之前已经移植了整个系统，这里只需要移植新的内核映像文件，如果读者是第一次移植则需要参考前面的章节将新生成的内核连同 Bootloader、文件系统一并下载到开发板上。

### 1．播放音频文件测试

使用文件系统中带的 madplay 播放 mp3 文件。播放命令为 madplay  音乐文件，播放过程如图 10.4 所示。在播放的过程中，将音响的输入线或者耳塞插入 mini2440 的音频 OUT 输出口。

```
# madplay zyfx.mp3
```

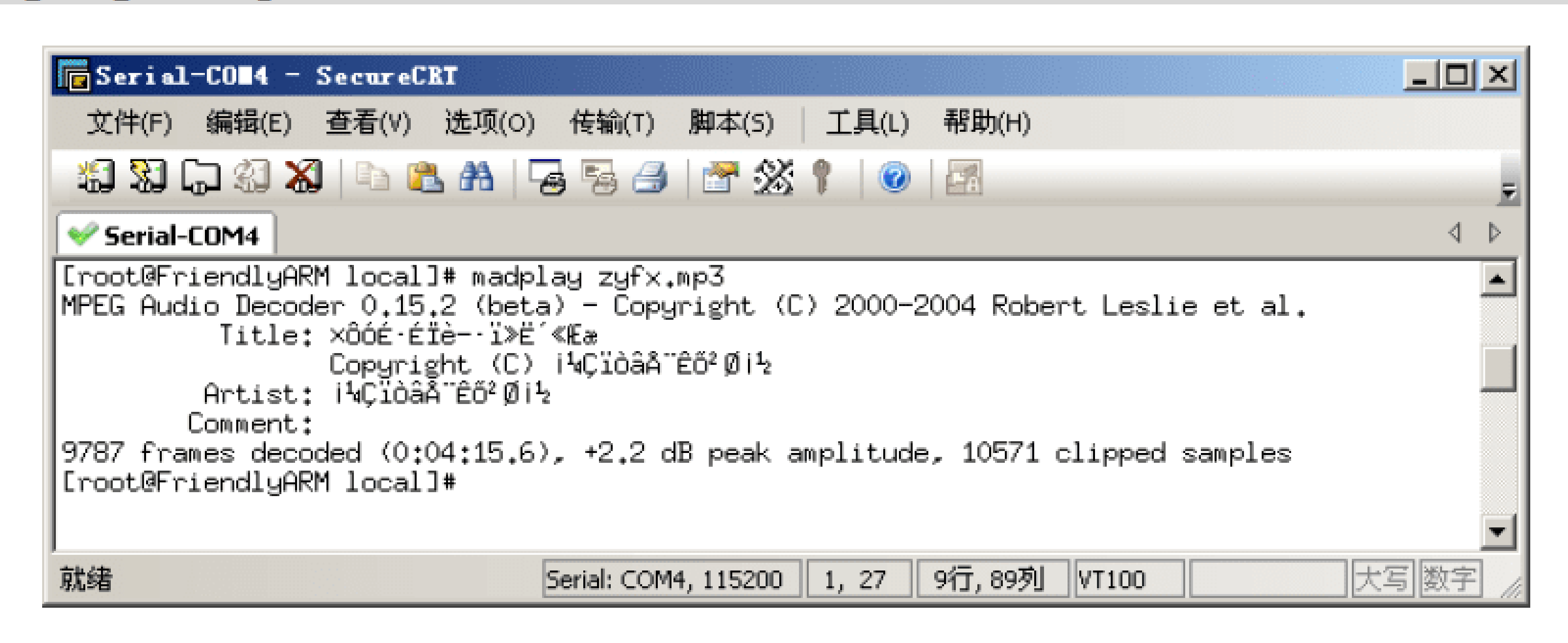



图 10.4　开发板上播放 mp3 文件

**注意：**如果读者的开发板中的文件系统不带 madplay 播放器，10.6 节将简单介绍 madplay 的移植过程。读者可以先移植 madplay，然后再测试 UDA1341 驱动的移植。

#### 2．录音测试

开发板文件系统自带录音程序 Recorder，将麦克风插入开发板的音频 MIC 输入口，运行该程序进行录音测试，该录音程序以图形界面的形式运行。在 FriendlyARM 标签页中双击运行 Recorder 程序，单击 REC 按钮进行录音，单击 STOP 按钮停止录音，单击 Play 按钮播放刚才的录音。

## 10.6　音频播放程序 madplay 的移植

前面已经详细介绍了音频驱动及其移植过程，本节将介绍运行在驱动和内核之上的播放器 madplay 的移植。

### 10.6.1　准备移植需要的源文件

移植 madplay 需要播放器代码、mp3 库文件和编码、解码库等。下面为移植需要准备相关源代码，读者可以到网上下载最新代码进行移植。

- madplay-0.15.2b.tar.gz：播放程序压缩包；
- libmad-0.15.1b.tar.gz：madplay 库文件；
- libid3tag-0.15.1b.tar.gz：针对 mp3 的解码库；
- zlib-1.2.3.tar.bz2：用于文件压缩和解压。

### 10.6.2　交叉编译

编译 madplay 时需要库文件的支持，因此先编译它依赖的库文件，然后再编译播放器。编译的过程包括 configure 生成 Makefile，设置安装路径，设置交叉编译工具，编译和安装等，与多数应用程序交叉编译过程类似。

#### 1．编译安装zlib-1.2.3.tar.bz2

解压文件 zlib-1.2.3.tar.bz2，进入解压目录，使用 configure 生成 Makefile，设置交叉编译工具，进行编译和安装，具体过程如下：

```
# tar zxvf zlib-1.2.3.tar.bz2
# cd zlib-1.2.3
# ./configure --prefix=/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc
```

修改 Makefile，将 x86 上的编译工具修改为交叉编译工具。

```
CC=arm-linux-gcc
```

```
LDSHARED=arm-linux-gcc
CPP=arm-linux-gcc -E
AR=arm-linux-ar rc
RANLIB=arm-linux-ranlib
TAR=arm-linux-tar
```

修改完 Makefile 后，使用 make 和 make install 进行编译和安装。

```
# make
# make install
```

### 2．编译安装libid3tag-0.15.1b.tar.gz

与编译 zlib-1.2.3.tar.bz2 的过程基本类似，首先解压文件 libid3tag-0.15.1b.tar.gz，进入解压目录，使用 configure 生成 Makefile，设置交叉编译工具，进行编译和安装。不同的是在 configure 命令中还指定了依赖的头文件和库文件路径，具体过程如下：

```
# tar zxvf libid3tag-0.15.1b.tar.gz
# cd libid3tag-0.15.1b
#  ./configure  --prefix=/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc
--host=arm-linux \
--enable-static --disable-shared \
CPPFLAGS=-I/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/include \
LDFLAGS=-L/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib
```

### 3．编译安装libmad-0.15.1b.tar.gz

交叉编译 libmad-0.15.1b.tar.gz 的过程和 libid3tag-0.15.1b.tar.gz 基本相同。编译的时候遇到编译选项的错误，通过下载补丁文件 libmad.patch，打上补丁后，编译安装通过。

```
# tar zxvf libmad-0.15.1b.tar.gz
# cd libmad-0.15.1b
# patch -p1<libmad.patch
#  ./configure  --prefix=/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc
--host=arm-linux \
--enable-static --disable-shared \
CPPFLAGS=-I/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/include \
LDFLAGS=-L/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib
```

### 4．编译安装madplay-0.15.2b.tar.gz

交叉编译安装 madplay-0.15.2b.tar.gz 的过程如下：

```
# tar zxvf madplay-0.15.2b.tar.gz
# cd madplay-0.15.2b
#  ./configure  --prefix=/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc
--host=arm-linux \
--enable-static --disable-shared \
CPPFLAGS=-I/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/include \
LDFLAGS=-L/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib
```

## 10.6.3　移植和测试

可执行文件 madplay 安装在目录/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/bin 下。

在交叉编译之前最好能在 X86 平台上编译通过，通过 ldd 查看 madplay 所依赖的库文件，将需要的对应/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib 下的库文件一并下载到开发板上。在开发板上执行测试。

```
# madplay zyfx.mp3
```

在开发板上播放音频文件结束后，打印播放歌曲的帧数、所花时间等信息。

```
9787 frames decoded (0:04:15.6), +2.2 dB peak amplitude, 10571 clipped
samples
```

### 10.6.4　编译中可能遇到的问题

在移植的过程中遇到几个典型的问题，这里列出来供读者参考。

（1）更换编译器时，可能会出现的问题。

ELF file OS ABI invalid

解决方法：库文件格式不对，可能是依赖了错误版本的库文件，清除掉库文件路径：

```
# export LD_LIBRARY_PATH=
```

（2）写 configure 参数时应该小心格式，格式错误时，可能有以下问题：

checking for suffix of object files... configure: error: cannot compute suffix of object files: cannot compile

解决方法：通过查看 config.log，ac_cv_env_CPPFLAGS_value 和 ac_cv_env_LDFLAGS_value 的设置有错。Configure 命令后的-I 和路径/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/include 之间不能存在空格–L 和路径/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib 之间不能有空格。

## 10.7　小　　结

本章重点理解音频驱动的代码，熟悉在配置内核时如何添加音频驱动 ALSA 框架和 OSS 框架，同时了解不同的音频接口编程，最后掌握针对 Mini2440 的音频驱动移植和音频播放器的移植过程。本章从底层驱动讲到内核配置、再到音频编程和音频应用程序移植，通过本章的学习，读者可以独立开发嵌入式音频系统。

# 第 11 章　SD 卡驱动移植

SD 卡（Secure Digital Memory Card），安全数码卡，是一种基于 Flash 的新一代存储设备，被广泛地用于便携式设备，例如移动电话、数码相机、个人数码助理（PDA）和多媒体播放器等。SD 卡拥有体积小，容量大、数据传输快、移动灵活及安全等优点。因其价格低廉，应用也越来越广泛，本章将重点介绍其驱动分析和移植过程。

## 11.1　SD 卡简介

SD 存储卡是专门为满足安全、大容量和内置于消费者的新型语音和视频电子设备中而设计的。SD 内存卡将包含的机械保护装置，遵循 SDMI 标准，具有安全、快速、大容量等特性。SD 卡的安全系统采用相互认证和“新密码算法”以防止卡中的内容被非法使用。下面将从以下几个方面简单介绍 SD 卡协议内容。

### 11.1.1　SD 卡系统概念

下面分别描述 SD 卡的读写特性、容量、速度、电压等特性和分类。

- 读写特性：根据读写特性可以将 SD 卡分为两种。一种为读/写卡，这种卡生产出来就是一张空白卡，专门用于记录用户的视频声音、图像的大容量记忆卡；另一种为只读卡，这种卡在制造时就定制了内容，其典型的应用是在软件、音频或视频等的发行媒体中。
- 支持电压：根据支持的电压可以将 SD 卡分为高电压 SD 卡和双重电压 SD 卡两类。
- 卡容量：根据卡的容量大小将 SD 分为两类型。一类为标准容量的 SD 卡，其支持的容量上线为 2GB，包括 2GB 在内；另一种为高容量 SD 卡，其容量超过 2GB，最大可达 32GB。
- 速度：根据速度类可以将 SD 卡分为 4 种速度类。类 0，这种类型卡兼具所有类型的优点；类 2，其速度大于等于 2MB/S；类 4，其速度大于等于 4MB/S；类 6，其速度大于等于 6MB/S。高容量 SD 卡支持速度类描述，其性能相当于或超过类 2。

### 11.1.2　SD 卡寄存器

每张卡都有一系列寄存器的信息，寄存器的信息如表 11.1 所示。

表 11.1　SD卡寄存器信息

| 名字 | 宽　度 | 描　述 |
|---|---|---|
| CID | 128 | 卡识别号，每张卡都有唯一的识别号 |
| RCA | 16 | 发布卡的地址，卡的局部系统地址，在初始化过程中，由主机和卡动态支持 |
| DSR | 16 | 驱动级寄存器，配置卡的驱动输出 |
| CSD | 128 | 卡的协议数据，关于卡的操作状态数据 |
| SCR | 64 | 卡配置寄存器，关于卡特性容量的信息 |
| OCR | 32 | 操作状态寄存器 |
| SSR | 512 | SD 状态，有关卡拥有的特性信息 |
| CSR | 32 | 卡状态，有关卡状态的信息 |

## 11.1.3　SD 功能描述

主机与卡之间的通信都是由主机控制的，主机发送的命令有两种类型，分别为广播命令和地址（点对点）命令。

- 广播命令：该命令是发给所有的卡，有些广播命令需要响应。
- 地址（点对点）命令：这些命令发往具体地址的卡，并且从这些卡生成响应。
- 卡识别模式：主机被复位或者在总线上寻找新卡时，主机处于该状态下。卡在复位以后和收到 SEND_RCA 命令以前都处于此模式下。
- 数据传输模式：卡在它们的 RCA 第一发布后进入数据传输模式。主机识别总线上所有的卡后进入数据传输模式。

下面通过表 11.2 说明卡的状态与操作模式之间的依赖关系，SD 的每种状态都关联一种操作模式，其状态图将在随后进行介绍。

表 11.2　卡的状态和操作模式的对应关系

| 卡　状　态 | 操 作 模 式 |
|---|---|
| 无活动状态 | 无活动 |
| 空闲态 | 卡识别模式 |
| 准备态 | |
| 识别态 | |
| 等待态 | 数据传输模式 |
| 传输态 | |
| 发送数据态 | |
| 接收数据态 | |
| 编程态 | |
| 断开态 | |

### 1．操作状态的验证

通过一系列过程后，主机才能识别卡。下面给出它们的通信过程。

- 在主机和卡通信前，主机不知道卡支持的电压，卡也不知道是否支持主机当前提供的电压。主机将发布一个复位命令（CMD0），带着它能提供给卡的电压信息。

- 为了验证 SD 卡的接口操作状态，主机发送 SEND_IF_COND(CMD8)，SD 卡通过分析 SEND_IF_COND 命令参数检查操作状态的有效性，主机通过检查 SD 分析后的响应来判断电压的有效性。
- 如果 SD 能够操作在提供的电压下，则发回的响应带上提供的电压，且检验模式被设置在命令参数中。如果 SD 卡不支持主机提供的电压，则不响应且保持在空闲态下。在发送 ACMD41 命令初始化高容量 SD 卡前，强制发送 CMD8 命令。
- 强制低电压主机在发送 CMD8 前发送 ACMD41。万一双重电压 SD 卡没有收到 CMD8 命令且工作在高电压状态，在这种情况下，低电压主机不发送 CMD8 命令给卡，则收到 ACMD41 后进入无活动状态。
- SD_SEND_OP_COND（ACMD41）命令是为 SD 卡主机识别卡或电压不匹配时拒绝卡的机制而设计的。主机发送命令操作数代表要求的电压窗口大小。如果 SD 卡在所给的范围内不能实现数据传输，将放弃下一步的总线操作而进入无活动状态。操作状态寄存器也将被定义。
- 在主机发出复位命令（CMD0）后，主机将先发送 CMD8 再发送 ACMD41 命令重新初始化 SD 卡。

卡的识别模式的状态可以用下面的状态图 11.1 表示。

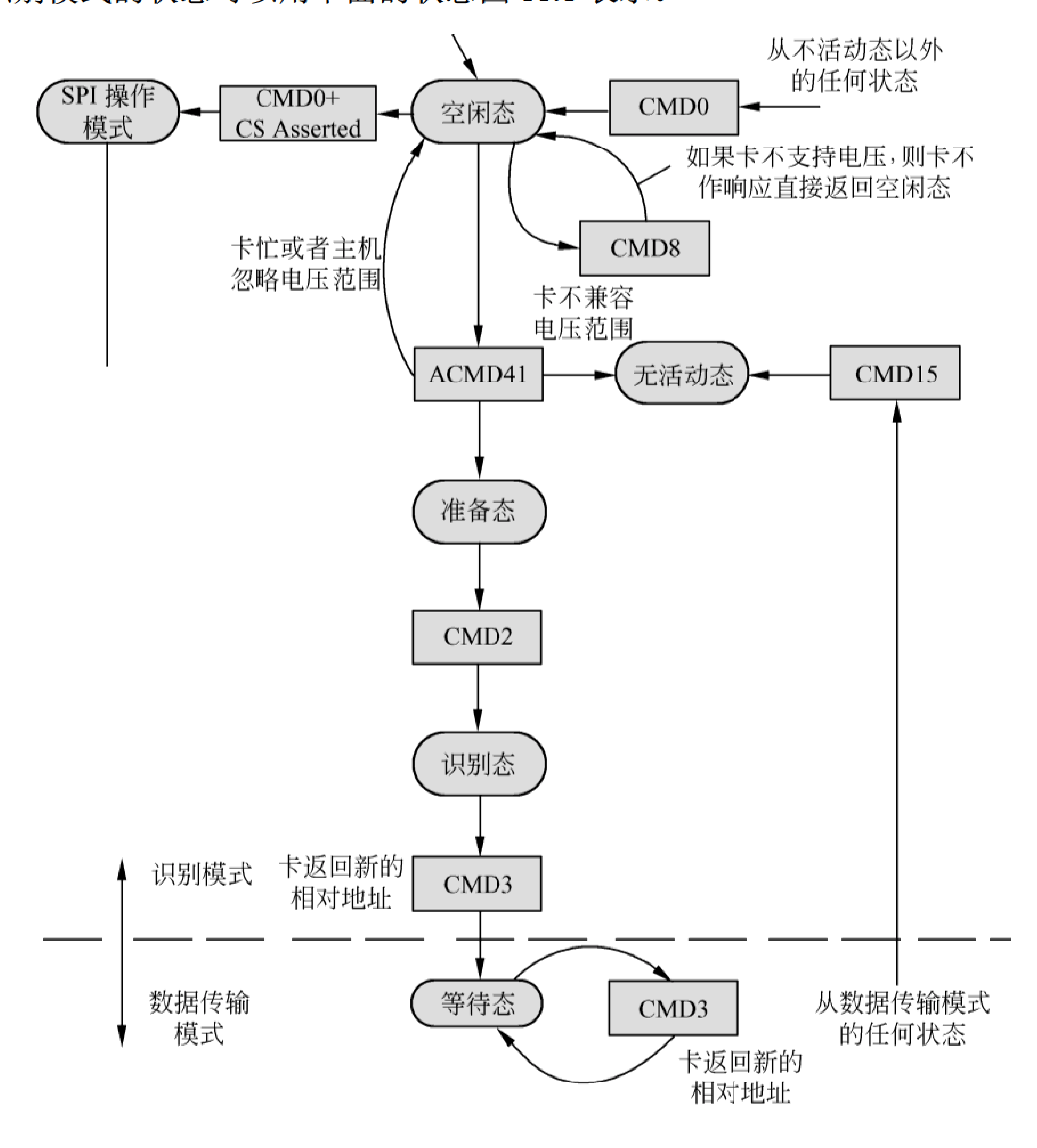



图 11.1　卡在识别模式下的命令流程

### 2．卡的初始化和识别处理

当总线被激活后，主机就开始卡的初始化和识别处理。初始化处理设置它的操作状态和设置 OCR 中的 HCS 比特位命令 SD_SEND_OP_COND（ACMD41）开始。HCS 比特位被设置为 1 表示主机支持高容量 SD 卡。HCS 被设置为 0 表示主机不支持高容量 SD 卡。

卡的初始化和识别更详细的流程，如图 11.2 所示。

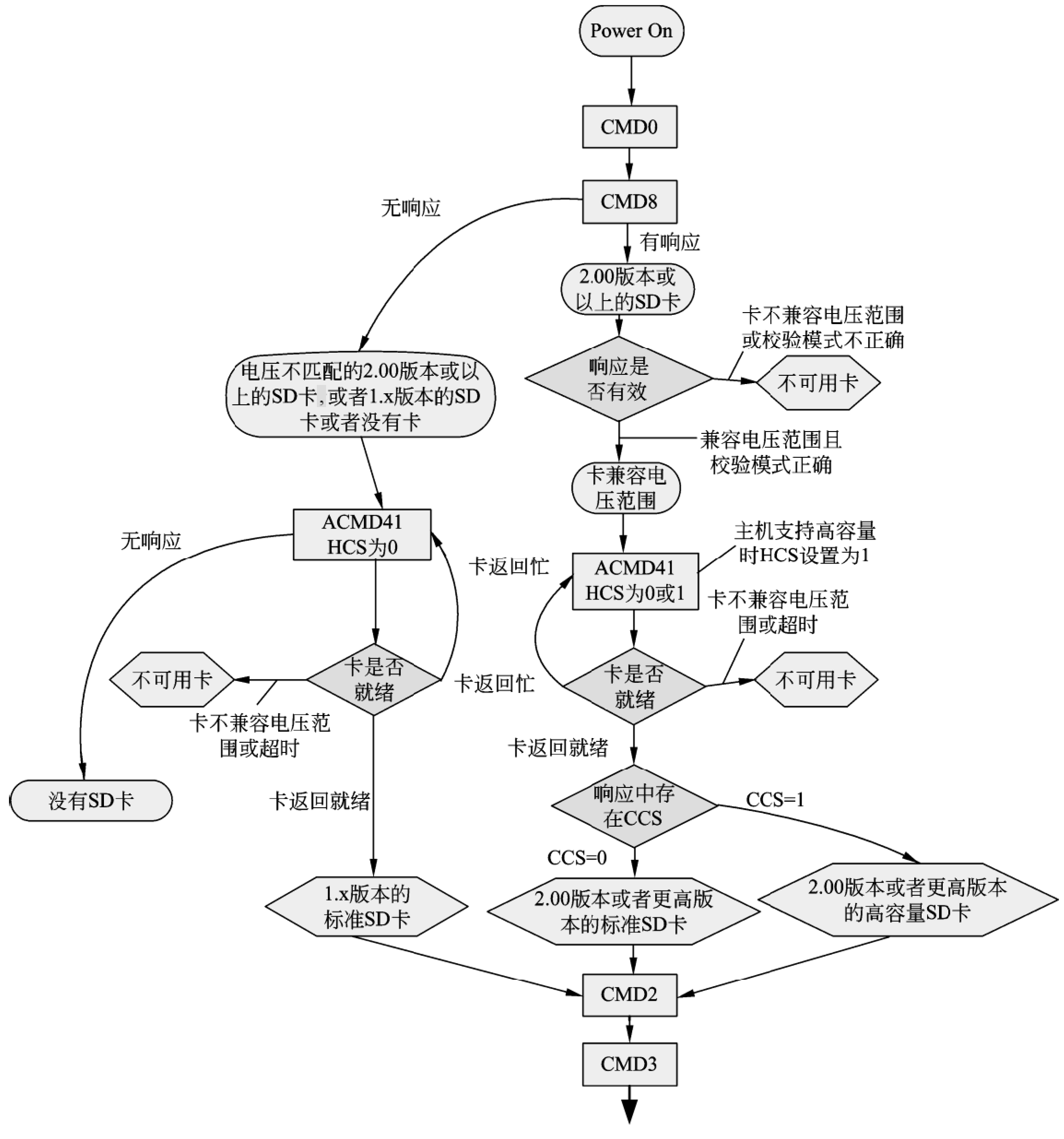



图 11.2　卡的初始化和识别流程

### 3．数据传输模式

卡的识别模式结束后，主机时钟 fpp（数据传输时钟速率）将保持为 $f_{OD}$（卡识别模式下的时钟），因为有些卡对操作时钟有限制。主机必须发送 SEND_CSD（CMD9）来获得

卡规格数据寄存器内容，如块大小、卡容量等。广播命令 SET_DSR（CMD4）配置所有识别卡的驱动阶段。它对 DSR 寄存器进行编程以适应应用总线布局、总线上的卡数目和数据传输频率。

SD 卡数据传输模式下的状态图，如图 11.3 所示。

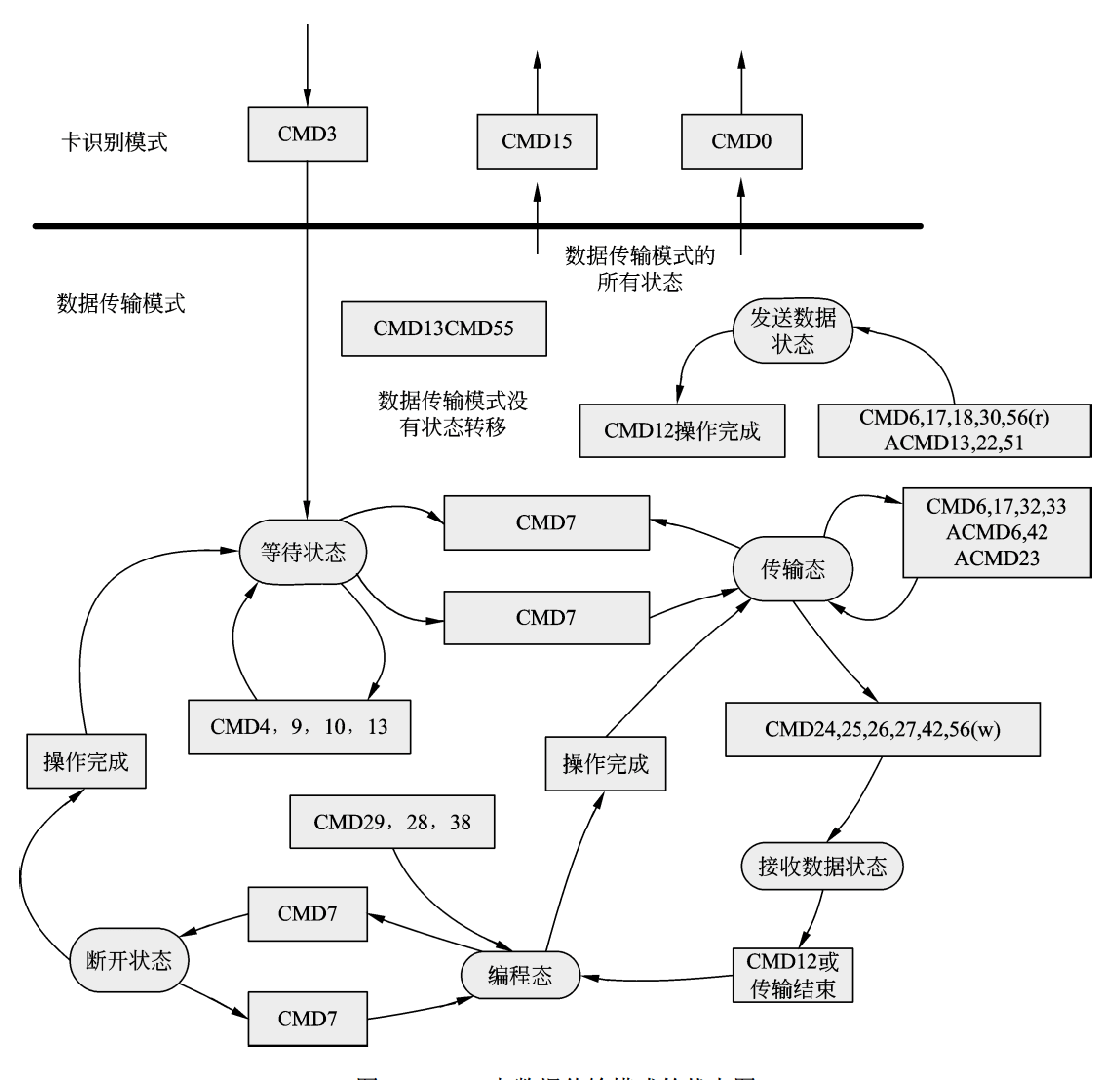



图 11.3　SD 卡数据传输模式的状态图

# 11.2　SD 卡驱动程序分析

SD 卡驱动程序包括驱动的注册和注销、设备接口函数和 I/O 操作。在 linux-2.6.29 内核 MMC 子系统中支持 SD 卡驱动。本节对 MMC 源码进行分析，后面将介绍 SD 卡驱动移植过程。MMC 子系统在 driver/mmc 目录下进行描述，该目录下包括 host、core、card 这 3 个文件夹，下面分别对这 3 个部分进行介绍。

## 11.2.1　host 驱动部分

host 驱动部分是针对不同类型主机的驱动，支持的开发板包括 atmel、S3C 等。这里就以 S3C 系统为例介绍 host 部分的主要内容。

### 1．驱动的注册函数

驱动的注册函数 s3cmci_init()，用于注册平台设备驱动。

```
static int   init s3cmci init(void)
{
    platform_driver_register(&s3cmci_2440_driver);  //注册平台设备驱动
    return 0;
}
```

### 2．驱动注销函数

驱动注销函数 s3cmci_exit()，用于注销平台设备驱动。

```
static void   exit s3cmci exit(void)
{
    platform_driver_unregister(&s3cmci_2440_driver);   //注销平台设备驱动
}
```

### 3．接口函数

平台设备接口函数包括 probe、remove、shutdown、suspend、resume。其结构如下：

```
static struct platform_driver s3cmci_2440_driver = {
    .driver.name     = "s3c2440-sdi",
    .driver.owner    = THIS_MODULE,
    .probe           = s3cmci_2440_probe,
    .remove          =   devexit p(s3cmci remove),
    .shutdown        = s3cmci shutdown,
    .suspend         = s3cmci suspend,
    .resume          = s3cmci_resume,
};
```

### 4．探针函数

探针函数 s3cmci_probe()，用于分配 s3cmci_host 结构体，然后对该结构体进行设置。对结构体 mmc_host 进行设置，将结构体 mmc 添加到主机。

```
static int __devinit s3cmci_probe(struct platform_device *pdev, int is2440)
{
    struct s3cmci_host  *host;
    struct mmc_host *mmc;
    int ret;
    /*为主机设备分配空间*/
    mmc = mmc_alloc_host(sizeof(struct s3cmci_host), &pdev->dev);
    if (!mmc) {
        ret = -ENOMEM;
        goto probe_out;
```

```
}
/*对 host 结构体各个字段进行设置*/
host = mmc_priv(mmc);
host->mmc        = mmc;
host->pdev       = pdev;
host->is2440     = is2440;
/*设置平台数据*/
host->pdata = pdev->dev.platform_data;
if (!host->pdata) {
    pdev->dev.platform_data = &s3cmci_def_pdata;
    host->pdata = &s3cmci_def_pdata;
}
/*初始化自旋锁，自旋锁在使用前应该被初始化*/
spin_lock_init(&host->complete_lock);
/*函数 tasklet_init ()用于初始化一个 tasklet，参数 pio_tasklet 是软中断响应函
数*/
tasklet_init(&host->pio_tasklet, pio_tasklet, (unsigned long) host);
/*结构体参数设置*/
host->sdiimsk   = S3C2440_SDIIMSK;
host->sdidata   = S3C2440_SDIDATA;
host->clk_div   = 1;
host->dodma         = 0;
host->complete_what     = COMPLETION_NONE;
host->pio_active    = XFER_NONE;
host->dma           = S3CMCI_DMA;
/*获取平台资源信息*/
host->mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
/*该函数的任务是检查申请的资源是否可用，如果可用则申请成功，并标志为已经使用，其他
驱动想再申请该资源时就会失败*/
host->mem = request_mem_region(host->mem->start,
                  RESSIZE(host->mem), pdev->name);
/*系统在运行时，外设的 I/O 内存资源的物理地址是已知的，由硬件的设计决定。但是 CPU
通常并没有为这些已知的外设 I/O 内存资源的物理地址预定义虚拟地址范围，驱动程序并不能
直接通过物理地址访问 I/O 内存资源，而必须将它们映射到内核虚地址空间内（采用页表），
然后才能根据映射所得到的内核虚地址范围，通过访内指令访问这些 I/O 内存资源。*/
host->base = ioremap(host->mem->start, RESSIZE(host->mem));
/*获取设备的中断号*/
host->irq = platform_get_irq(pdev, 0);
if (host->irq == 0) {
    dev_err(&pdev->dev, "failed to get interrupt resouce.\n");
    ret = -EINVAL;
    goto probe_iounmap;
}
/*向系统申请中断*/
if (request_irq(host->irq, s3cmci_irq, 0, DRIVER_NAME, host)) {
    dev_err(&pdev->dev, "failed to request mci interrupt.\n");
    ret = -ENOENT;
    goto probe_iounmap;
}
/*关闭中断*/
disable_irq(host->irq);
/*给定端口号转换为中断号*/
host->irq_cd = s3c2410_gpio_getirq(host->pdata->gpio_detect);
/*添加 irq_cd 的中断号为 IRQ_EINT16 且设置 GPG8 脚为 16 号中断的输入引脚*/
host->irq_cd = IRQ_EINT16;
s3c2410_gpio_cfgpin(S3C2410_GPG8, S3C2410_GPG8_EINT16);
/*获取 dma 通道的控制权*/
```

```
    s3c2410_dma_request(S3CMCI_DMA, &s3cmci_dma_client, NULL);
    /*获取时钟响应给时钟的生产者 producer()*/
    host->clk = clk_get(&pdev->dev, "sdi");
    /*当时钟源运行的时候通知系统，参数 host->clk 为时钟源*/
    clk_enable(host->clk);
    /*获得当前时钟频率*/
    host->clk_rate = clk_get_rate(host->clk);
    /*下面是对 mmc 结构体参数的设置*/
    mmc->ops    = &s3cmci_ops;
    mmc->ocr_avail  = MMC_VDD_32_33 | MMC_VDD_33_34;
    mmc->caps   = MMC_CAP_4_BIT_DATA;
    mmc->f_min  = host->clk_rate / (host->clk_div * 256);
    mmc->f_max  = host->clk_rate / host->clk_div;
    if (host->pdata->ocr_avail)
        mmc->ocr_avail = host->pdata->ocr_avail;
    mmc->max_blk_count  = 4095;
    mmc->max_blk_size = 4095;
    mmc->max_req_size = 4095 * 512;
    mmc->max_seg_size   = mmc->max_req_size;

    mmc->max_phys_segs  = 128;
    mmc->max_hw_segs    = 128;
    /*注册带 CPU 频率的 host 驱动*/
    s3cmci_cpufreq_register(host);
    /*初始化 mmc*/
    mmc_add_host(mmc);
    /*设置驱动数据*/
    platform_set_drvdata(pdev, mmc);
    return 0;
}
```

### 5. mmc接口函数

mmc 子系统的接口函数包括 request、set_ios、get_ro、get_cds。其结构如下：

```
static struct mmc_host_ops s3cmci_ops = {
    .request    = s3cmci_request,       //实现命令和数据的发送
    .set_ios    = s3cmci_set_ios,       //根据核心层传来的 ios 来设置硬件 IO
    .get_ro     = s3cmci_get_ro,        //从 GPIO 口读取，判断卡是否写保护
    .get_cd     = s3cmci_card_present,  //从 GPIO 口读取，判断卡是否存在
};
```

### 6. 传递结构体为mmc_request类型的请求

函数 s3cmci_request()用于 CORE 部分发送 mrq 请求。

```
static void s3cmci_request(struct mmc_host *mmc, struct mmc_request *mrq)
{
    struct s3cmci_host *host = mmc_priv(mmc);

    host->status = "mmc request";
    host->cmd_is_stop = 0;
    host->mrq = mrq;
/*如果卡准备就绪，则通过 s3cmci_send_request ()发送请求，将 mrq 赋给 host->mrq，如
果卡没有准备就绪，则调用 mmc_request_done ()终止请求*/
    if (s3cmci_card_present(mmc) == 0) {
```

```
        dbg(host, dbg_err, "%s: no medium present\n", __func__);
        host->mrq->cmd->error = -ENOMEDIUM;
        mmc_request_done(mmc, mrq);
    } else
/*函数 s3cmci_send_request ()首先判断是否为发送数据命令，如果为发送数据则通过函数 s3cmci_send_request ()建立数据，然后判断是否为 dma 方式，如果为 dma 方式则通过 dma 方式发送数据，否则采用 fifo 方式发送数据。如果为命令则通过函数 s3cmci_send_command()发送命令*/
        s3cmci_send_request(mmc);
}
```

## 11.2.2　core 驱动部分

core 驱动部分完成不同协议和规范的实现，包括设置在 11.1 节中介绍的有关 SD 卡相关的状态或修改状态、修改寄存器等操作。

### 1. 用于卡的探测和初始化函数mmc_sd_init_card()

在重启时，函数 mmc_sd_init_card()参数 oldcard 中包含准备初始化的卡，该函数检测卡的有效性，并对该卡初始化。该函数首先让卡的状态回到空闲态，然后设置操作状态寄存器，接着进行 SD 卡主机识别或电压匹配，正确识别和匹配后，读取卡的识别号；比较读取的 CID 与原来的 CID 是否相同，不相同则需要重新为卡分配结构体；最后对卡进行设置和初始化。

```
static int mmc_sd_init_card(struct mmc_host *host, u32 ocr, struct mmc_card *oldcard)
{
    struct mmc_card *card;
    int err;
    u32 cid[4];
    unsigned int max_dtr;
    BUG_ON(!host);
    WARN_ON(!host->claimed);
    /*改变状态寄存器 OCR 的值时，需要卡的状态回到空闲态。等待 1ms 让卡响应*/
    mmc_go_idle(host);
    /*SD_SEND_IF_COND 是用于验证 SD 卡接口操作状态的有效性命令（CMD8）。如果 SD_SEND_IF_COND 指示为符合 SD2.0 标准的卡，则设置操作状态寄存器 ocrbit30 指示能够处理块地址 SDHC 卡*/
    err = mmc_send_if_cond(host, ocr);
    if (!err)
        ocr |= 1 << 30;
    /*SD_SEND_OP_COND（ACMD41）该命令为 SD 卡主机识别或电压不匹配时拒绝机制而设计*/
    mmc_send_app_op_cond(host, ocr, NULL);
    /*如果主机采用 SPI 总线则采用适当的 CRC*/
    if (mmc_host_is_spi(host)) {
        mmc_spi_set_crc(host, use_spi_crc);
    }
    /*从卡中读取 CID，卡的识别号*/
    if (mmc_host_is_spi(host))
        mmc_send_cid(host, cid);
    else
        mmc_all_send_cid(host, cid);
```

```
    /*比较读取的 CID 与原来的 CID 是否相同*/
    if (oldcard) {
        if (memcmp(cid, oldcard->raw_cid, sizeof(cid)) != 0) {
            err = -ENOENT;
            goto err;
        }
        card = oldcard;
    } else {
        /*为卡分配结构体*/
        card = mmc_alloc_card(host, &sd_type);
        if (IS_ERR(card)) {
            err = PTR_ERR(card);
            goto err;
        }
        /*设置卡的类型*/
        card->type = MMC_TYPE_SD;
        memcpy(card->raw_cid, cid, sizeof(card->raw_cid));
    }
    if (!mmc_host_is_spi(host)) {
        /*获得卡的 RCA，该寄存器表示发布卡的地址，卡的局部系统地址，在初始化过程中，
          由主机和卡动态支持*/
        mmc_send_relative_addr(host, &card->rca);
        /*设置总线模式*/
        mmc_set_bus_mode(host, MMC_BUSMODE_PUSHPULL);
    }
    if (!oldcard) {
        /*获得卡 CSD，该寄存器表示卡的协议数据，关于卡的操作状态数据*/
        mmc_send_csd(card, card->raw_csd);
        /*卡的 CSD 结构的解码*/
        mmc_decode_csd(card);
        /*卡的 CID 结构解码*/
        mmc_decode_cid(card);
    }
    if (!mmc_host_is_spi(host)) {
        /*选择卡，后续的命令都依赖该操作*/
        mmc_select_card(card);
    }
    if (!oldcard) {
        /*获得卡的 SCR，该寄存器表示卡配置寄存器，关于卡特性容量的信息*/
        mmc_app_send_scr(card, card->raw_scr);
        /*解码 SCR 结构*/
        mmc_decode_scr(card);
        /*获得卡的 switch 信息*/
        mmc_read_switch(card);
    }
    /*尝试转化为高速*/
    mmc_switch_hs(card);
    /*计算总线速率*/
    max_dtr = (unsigned int)-1;
    if (mmc_card_highspeed(card)) {
        if (max_dtr > card->sw_caps.hs_max_dtr)
            max_dtr = card->sw_caps.hs_max_dtr;
    } else if (max_dtr > card->csd.max_dtr) {
        max_dtr = card->csd.max_dtr;
    }
    /*设置可能的最高主机的时钟*/
    mmc_set_clock(host, max_dtr);
```

```
    /*如果支持，转化为更宽的总线*/
    if ((host->caps & MMC_CAP_4_BIT_DATA) &&
        (card->scr.bus_widths & SD_SCR_BUS_WIDTH_4)) {
        err = mmc_app_set_bus_width(card, MMC_BUS_WIDTH_4);
        if (err)
            goto free_card;

        mmc_set_bus_width(host, MMC_BUS_WIDTH_4);
    }
    /*检查卡只读是否激活*/
    if (!oldcard) {
        if (!host->ops->get_ro || host->ops->get_ro(host) < 0) {
            printk(KERN_WARNING "%s: host does not "
                "support reading read-only "
                "switch. assuming write-enable.\n",
                mmc_hostname(host));
        } else {
            if (host->ops->get_ro(host) > 0)
                mmc_card_set_readonly(card);
        }
    }
    if (!oldcard)
        host->card = card;
    return 0;
/*注销 mmc 卡*/
free_card:
    if (!oldcard)
        mmc_remove_card(card);
err:
    return err;
}
```

### 2．删除SD卡函数mmc_sd_remove()

函数 mmc_sd_remove()用于移除主机 host，释放当前卡。

```
static void mmc_sd_remove(struct mmc_host *host)
{
    BUG_ON(!host);
    BUG_ON(!host->card);

    mmc_remove_card(host->card);
    host->card = NULL;
}
```

### 3．初始化主机结构体函数mmc_alloc_host()

函数 mmc_alloc_host()在 host 层探针函数中被调用，该函数为主机结构体分配空间，并初始化主机结构体。

```
struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
{
    /*为主机结构体分配空间*/
    host = kzalloc(sizeof(struct mmc_host) + extra, GFP_KERNEL);
    spin_lock(&mmc_host_lock);
    /*分配新的 idr 入口*/
    idr_get_new(&mmc_host_idr, host, &host->index);
```

```
    spin_unlock(&mmc_host_lock);
    /*设置设备的名字，对 host 设备相关字段进行设置*/
    dev_set_name(&host->class_dev, "mmc%d", host->index);
    host->parent = dev;
    host->class_dev.parent = dev;
    host->class_dev.class = &mmc_host_class;
    /*函数 device_initialize()用于初始化设备结构体，该函数一般为其它层做准备，这里
    是为 host 层作准备*/
    device_initialize(&host->class_dev);
    spin_lock_init(&host->lock);
    init_waitqueue_head(&host->wq);
    /*初始化一个工作队列*/
    INIT_DELAYED_WORK(&host->detect, mmc_rescan);
    /*初始化主机结构体的默认配置*/
    host->max_hw_segs = 1;
    host->max_phys_segs = 1;
    host->max_seg_size = PAGE_CACHE_SIZE;
    host->max_req_size = PAGE_CACHE_SIZE;
    host->max_blk_size = 512;
    host->max_blk_count = PAGE_CACHE_SIZE / 512;

    return host;
}
```

### 4．初始化主机硬件函数mmc_add_host()

函数 mmc_add_host()用于增加设备类，并启动 host。

```
int mmc_add_host(struct mmc_host *host)
{
    led_trigger_register_simple(dev_name(&host->class_dev), &host->led);
    device_add(&host->class_dev);
    mmc_start_host(host);
}
```

### 5．删除host硬件函数mmc_remove_host()

函数 mmc_remove_host()与函数 mmc_add_host()相对应，用于停止函数 mmc_add_host()启动的 host，删除设备类，注销 LED trigger。

```
void mmc_remove_host(struct mmc_host *host)
{
    mmc_stop_host(host);
    device_del(&host->class_dev);
    led_trigger_unregister_simple(host->led);
}
```

### 6．释放主机结构体函数mmc_free_host()

函数 mmc_free_host()与函数 mmc_alloc_host()对应，释放函数 mmc_alloc_host()初始化的 host 结构体。

```
void mmc_free_host(struct mmc_host *host)
{
    spin_lock(&mmc_host_lock);
    idr_remove(&mmc_host_idr, host->index);
```

```
    spin_unlock(&mmc_host_lock);
    put_device(&host->class_dev);
}
```

通过对 core 层函数进行分析，可以看出 core 层除了完成协议描述的部分外，还为 host 层提供了接口函数。

## 11.2.3　card 驱动部分

SD 卡属于块设备，card 驱动部分为了将 SD 卡驱动成为块设备。介绍该部分的内容时先介绍驱动结构体和接口函数结构体，然后介绍几个关键的驱动函数。

### 1．驱动的结构体mmc_driver

该结构体定义驱动的名字，驱动探针函数、驱动移除函数、驱动阻塞和驱动重启等函数。

```
static struct mmc_driver mmc_driver = {
    .drv            = {
        .name       = "mmcblk",
    },
    .probe          = mmc_blk_probe,
    .remove         = mmc_blk_remove,
    .suspend        = mmc_blk_suspend,
    .resume         = mmc_blk_resume,
};
```

### 2．块设备操作结构体mmc_bdops

结构体 mmc_bdops 中定义了块设备操作的接口函数。

```
static struct block_device_operations mmc_bdops = {
    .open           = mmc_blk_open,
    .release            = mmc_blk_release,
    .getgeo         = mmc_blk_getgeo,
    .owner          = THIS_MODULE,
};
```

### 3．块设备探针函数mmc_blk_probe()

该函数主要完成检验卡支持的命令，分配 mmc_blk_data 结构体空间，设置块的大小，最后设置 card 的 driver_data 字段，并注册 mmc 信息到系统。

```
static int mmc_blk_probe(struct mmc_card *card)
{
    struct mmc_blk_data *md;
    int err;
    char cap_str[10];
    /*检查卡支持的命令*/
    if (!(card->csd.cmdclass & CCC_BLOCK_READ))
        return -ENODEV;
    /*为 card 分配 mmc_blk_data 结构体空间*/
    md  =  mmc_blk_alloc(card);
```

```
    /*设置块的大小*/
    mmc_blk_set_blksize(md, card);
    /*将 md 设置为 card 的 driver_data 字段*/
    mmc_set_drvdata(card, md);
    /*把 mmc 包含的信息向系统进行注册，注册成功后就可以在文件系统对应目录下找到
    mmc_card 对应的结点设备*/
    add_disk(md->disk);
    return 0;
}
```

### 4．驱动的入口函数mmc_blk_init()

加载驱动时该函数被调用，该函数向内核申请注册一个块设备，然后进入核心层进行注册。

```
static int __init mmc_blk_init(void)
{
    /*向内核申请注册一个块设备*/
    res = register_blkdev(MMC_BLOCK_MAJOR, "mmc");
    /*进入核心层进行注册*/
    mmc_register_driver(&mmc_driver);
    return 0;
}
```

### 5．为块设备分配空间函数mmc_blk_alloc()

函数 mmc_blk_alloc()为块设备分配空间，并初始化一个请求队列，设置设备队列的 sector 大小。

```
static struct mmc_blk_data *mmc_blk_alloc(struct mmc_card *card)
{
    struct mmc_blk_data *md;
    int devidx, ret;
    /*在内存中查找第一个被清理过的 bit*/
    devidx = find_first_zero_bit(dev_use, MMC_NUM_MINORS);
    if (devidx >= MMC_NUM_MINORS)
        return ERR_PTR(-ENOSPC);
    /*从地址 dev_use 开始设置 bit，设置为 devidx*/
      set_bit(devidx, dev_use);
    /*分配结构体 mmc_blk_data 空间并初始化*/
    md = kzalloc(sizeof(struct mmc_blk_data), GFP_KERNEL);
    /*设置卡的状态为只读*/
    md->read_only = mmc_blk_readonly(card);
    /*分配设备的次设备号为 8*/
    md->disk = alloc_disk(1 << MMC_SHIFT);
    }
    spin_lock_init(&md->lock);
    md->usage = 1;
    /*初始化一个请求队列，并将该队列与卡关联*/
    mmc_init_queue(&md->queue, card, &md->lock);
    /*注册 mmc_blk_issue_rq 到 md->queue，当 md->queue 上有 request 待处理时，
    mmc_blk_issue_rq 就会被调用*/
    md->queue.issue_fn = mmc_blk_issue_rq;
    md->queue.data = md;
    /*注册相关的 mmc_blk _data 包含的块设备区*/
```

```
    md->disk->major     = MMC_BLOCK_MAJOR;
    md->disk->first_minor = devidx << MMC_SHIFT;
    md->disk->fops = &mmc_bdops;
    md->disk->private_data = md;
    md->disk->queue = md->queue.queue;
    md->disk->driverfs_dev = &card->dev;
    sprintf(md->disk->disk_name, "mmcblk%d", devidx);
    /*设置传输 sector 大小*/
    blk_queue_hardsect_size(md->queue.queue, 512);
    /*根据卡的类型设置容量*/
    if (!mmc_card_sd(card) && mmc_card_blockaddr(card)) {
        /*
         * The EXT_CSD sector count is in number or 512 byte
         * sectors.
        */
        set_capacity(md->disk, card->ext_csd.sectors);
    } else {
        /*
         * The CSD capacity field is in units of read_blkbits.
         * set_capacity takes units of 512 bytes.
        */
        set_capacity(md->disk,
            card->csd.capacity << (card->csd.read_blkbits - 9));
    }
    return md;
}
```

在该驱动部分还包括一些对块操作的函数，如 mmc_blk_open()、mmc_blk_get()、mmc_blk_put()、mmc_blk_release()和 mmc_blk_getgeo()等。

# 11.3　SD 卡移植步骤

SD 卡的驱动程序已经包含在内核中，只需要在编译内核时配置对 SD 卡驱动支持。这里使用的开发板为 mini2440，其对应的初始化文件为 mach-mini2440.c。

## 11.3.1　添加延时和中断

文件 mach-mini2440.c 可以参考 mach-smdk2440.c 进行修改，这里不对该文件的修改进行介绍，可以下载到开发板公司修改好的压缩文件。另外，延时和中断也在附带的代码中添加，添加延时在 s3cmci.c 文件中。

### 1．添加延时

修改 drivers/mmc/host/s3cmci.c 文件，在底层函数 pio_tasklet()中添加延时。

```
#include <linux/delay.h>
static void pio_tasklet(unsigned long data)
{
    struct s3cmci_host *host = (struct s3cmci_host *) data;
    disable_irq(host->irq);
    udelay(50);
}
```

### 2．添加中断设置

在函数 static int __devinit s3cmci_probe(struct platform_device *pdev, int is2440)中添加中断设置。粗体部分为添加部分。

```
host->irq_cd = s3c2410_gpio_getirq(host->pdata->gpio_detect);
host->irq_cd = IRQ_EINT16;
s3c2410_gpio_cfgpin(S3C2410_GPG8, S3C2410_GPG8_EINT16);
```

## 11.3.2　配置内核

在编译内核前需要对内核进行配置，让内核支持 SD 卡的访问。使用 make menuconfig 命令进入窗口配置界面，进入 Device Drivers 配置界面配置 MMC/SD/SDIO card support，如图 11.4 所示。

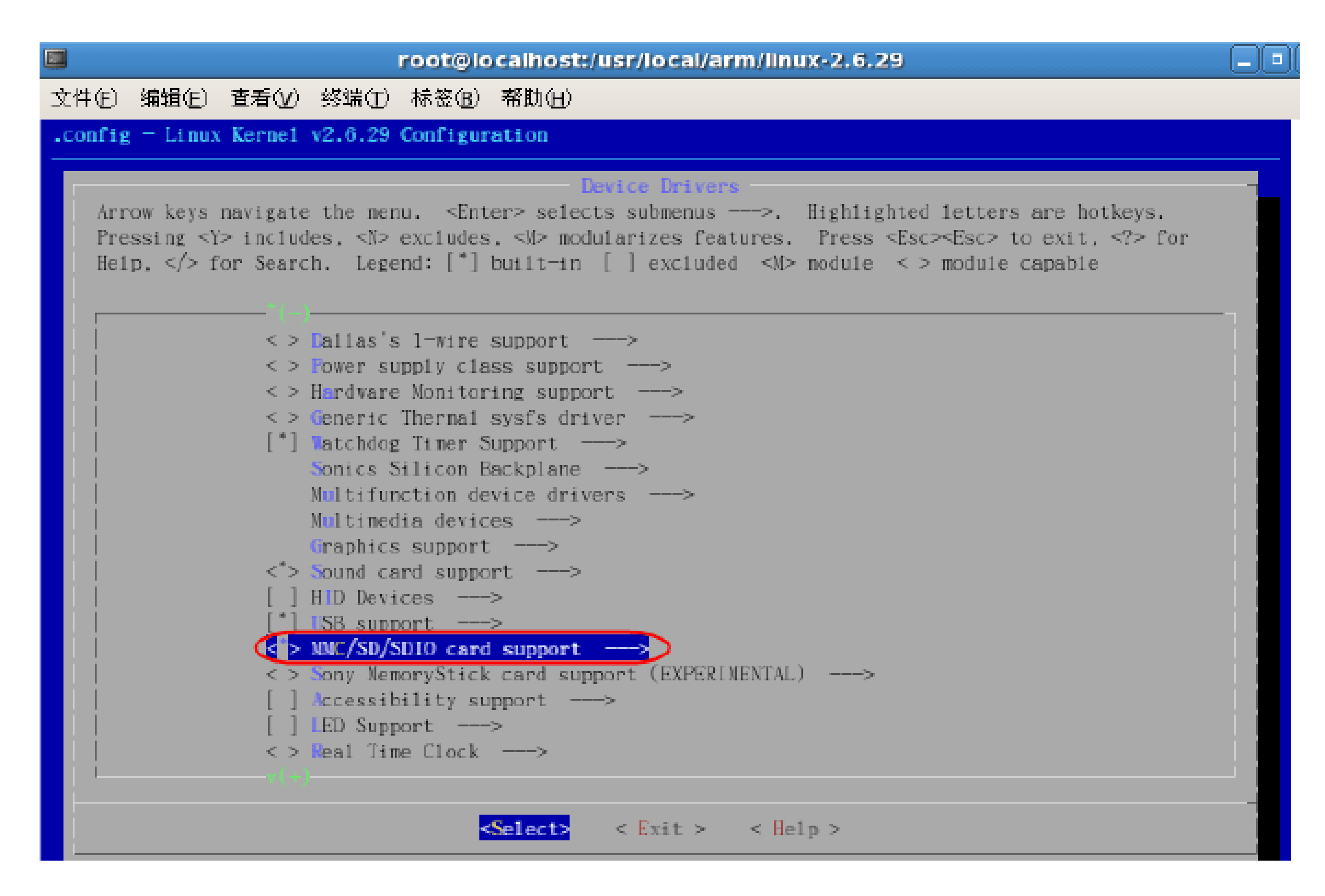



图 11.4　配置 MMC/SD/SDIO card support

选择对平台的支持，进入 MMC/SD/SDIO card support 配置窗口选择配置 Samsung S3C SD/MMC Card Interface support，如图 11.5 所示。选择该项后，内核支持 S3C 系列的 SD 卡驱动。

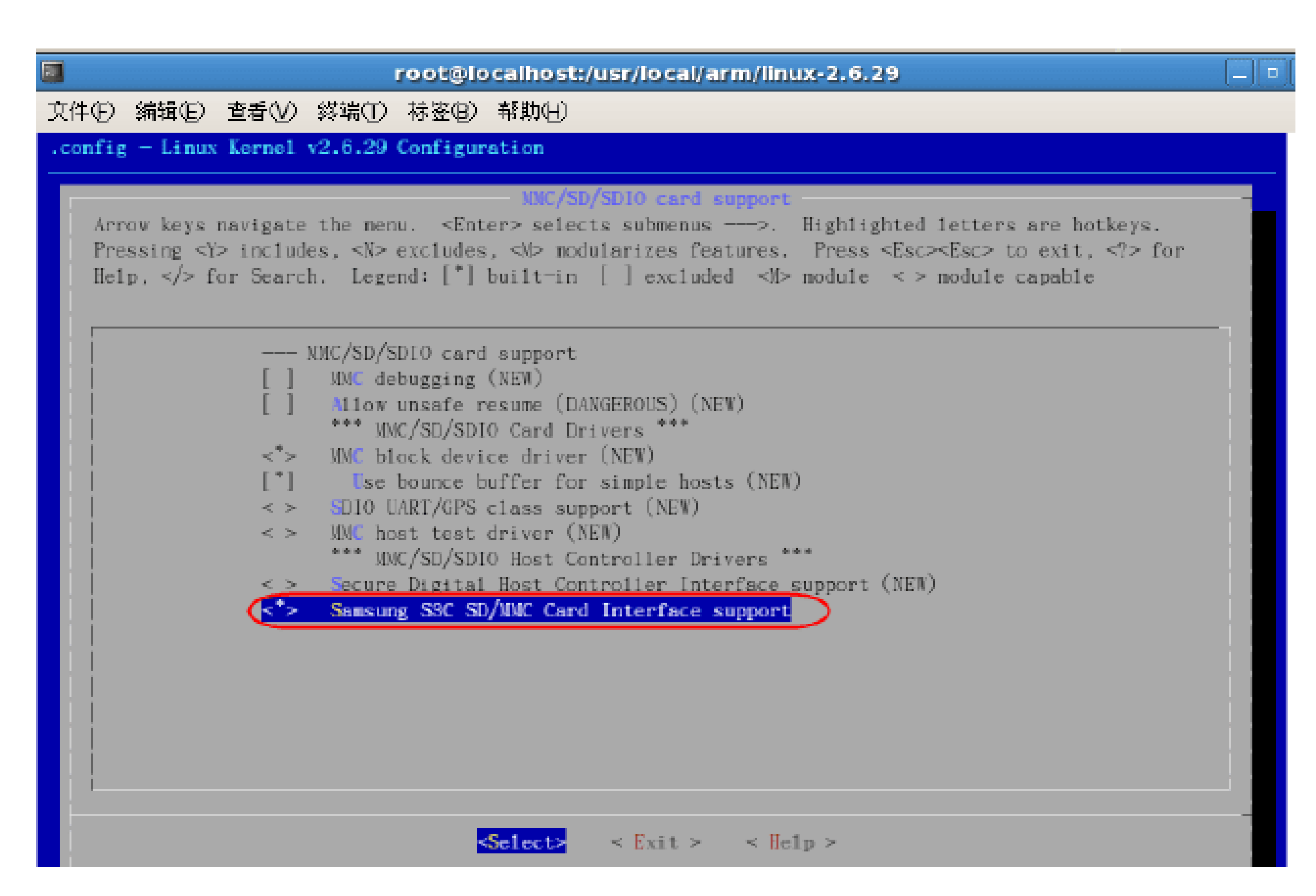



图 11.5　配置 Samsung S3C SD/MMC Card Interface support

保存配置后，使用下面的编译命令编译新内核映像文件，命令如下：

```
#make clean
#make zImage
```

## 11.3.3　烧写新内核

烧写新生成的内核映像文件时，系统的其他部分不必重新烧写。新内核烧写完成后，重新启动开发板，然后准备 SD 卡和扩展卡。将 SD 卡插入扩展卡，然后将扩展卡插入开发板 SD 卡插槽中，出现下面提示信息，如图 11.6 所示。

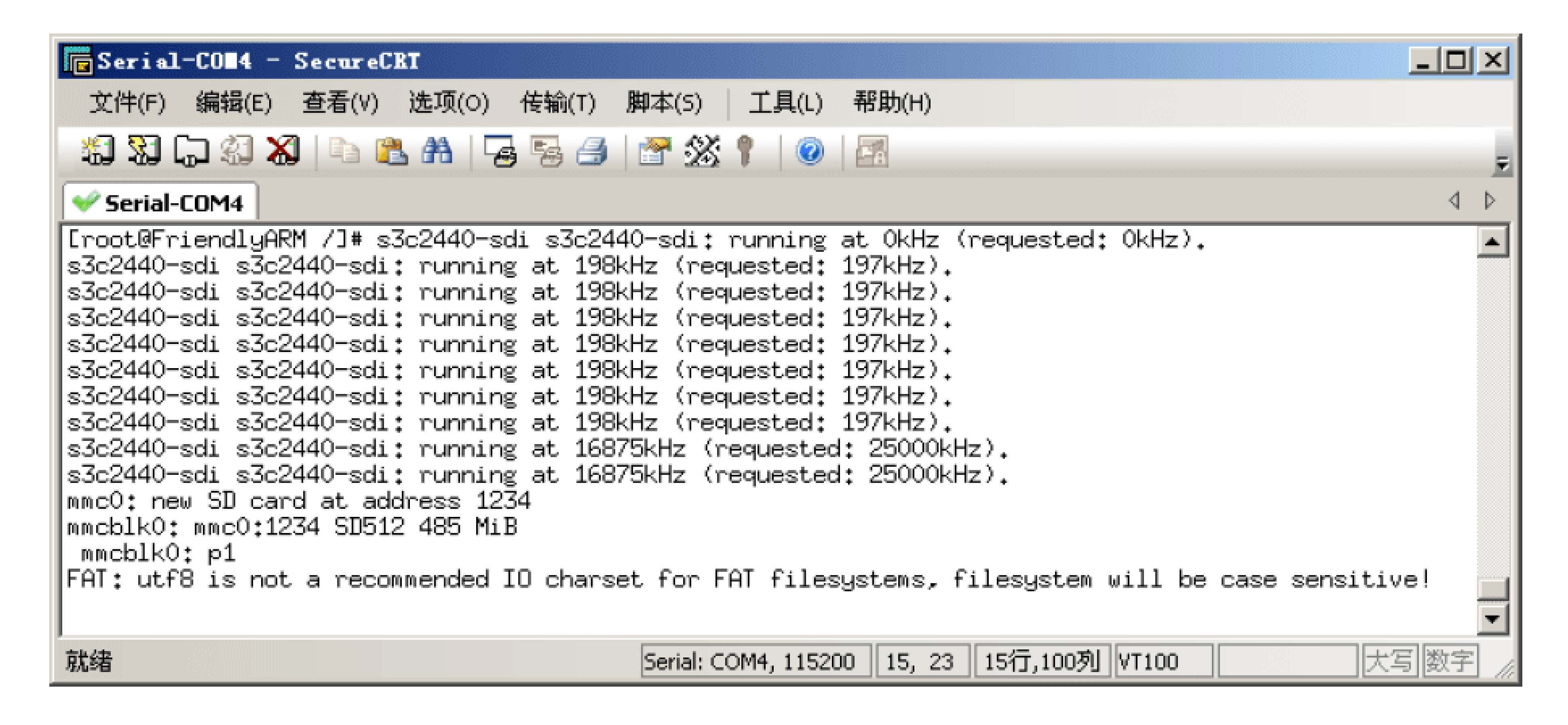



图 11.6　识别到 SD 卡信息

使用 ls /dev 命令可以查看在目录下多了设备结点 sdcard，如图 11.7 所示。

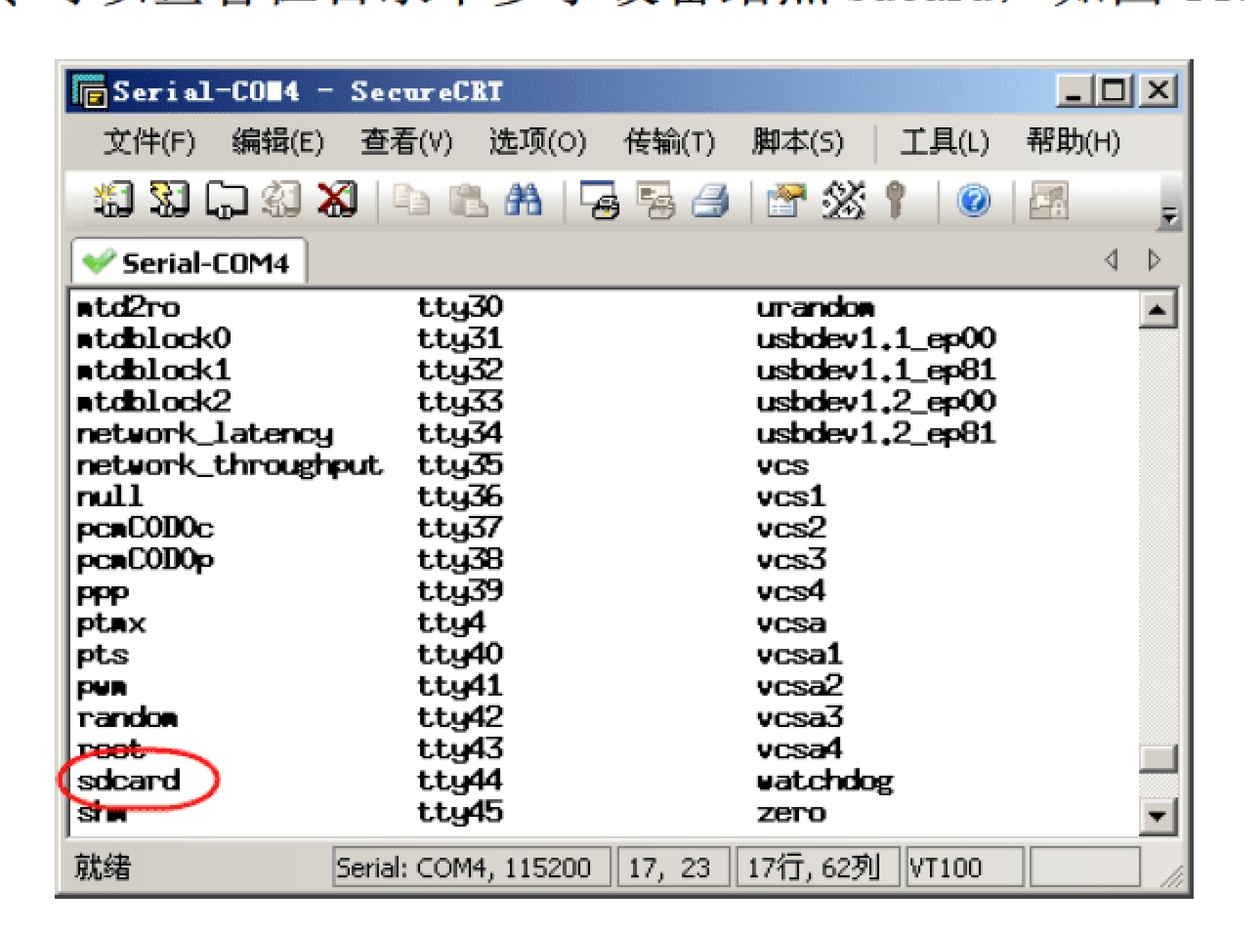



图 11.7　sdcard 结点

挂载该设备结点后，可以查看该设备内的信息，挂载前新建目录/mnt/sdcard，挂载命令和结果如下：

```
# mkdir /mnt/sdcard
# mount /dev/sdcard /mnt/sdcard
# ls /mnt/sdcard/
@samsung.ess         My Music           audio_play_list.txt
```

```
Audio                Other files          bootex.log
Ebook                Photos               software
Images               Sounds
Music                Videos
```

## 11.4　小　　结

本章重点为 SD 卡协议介绍和 SD 卡驱动分析，后面也介绍了 SD 卡驱动移植过程。随着 SD 卡存储容量增加和价格下降，其应用越来越广泛，SD 卡驱动在嵌入式系统中也将会受到关注。Linux 内核已经对 SD 卡驱动进行了支持，移植到其他平台时只需要做少量的修改即可。

# 第 12 章　NandFlash 驱动移植

在很多章节中都会涉及 NandFlash 的相关知识，比如 U-boot 中涉及对 NandFlash 的支持、在文件系统中涉及 NandFlash 支持的文件系统，因此本章在简单介绍 NandFlash 工作原理后，直接介绍其移植方法。NandFlash 相比 NorFlash 有很多优势，但需要驱动支持，本章将针对 Mini2440 讲解 NandFlash 的驱动移植过程。

## 12.1　NandFlash 介绍

对 NandFlash 存储芯片进行操作，必须通过 NandFlash 控制器才能完成，而不能通过对 NandFlash 进行总线操作。对于 NandFlash 的写操作只能以块方式进行写，对于 NandFlash 的读操作可以按字节进行读。

### 12.1.1　NandFlash 命令介绍

NandFlash 命令的执行过程是通过将命令发送到 NandFlash 控制器的命令寄存器来执行的。其命令的执行是分周期的，每条命令有一个或多个执行周期，每个执行周期有相应代码表示该周期将要执行的动作。NandFlash 命令主要包括 Read1、Read2、Read ID、Reset、Page Program、Block Erase、Read Status 等。

1．Read1

- 命令功能：表示将要读取 NandFlash 存储空间中一页的前半部分，且将内置指针定位到前半部分的第一个字节。
- 命令代码：00h。

2．Read2

- 命令功能：表示将要读取 NandFlash 存储空间中一页的后半部分，且将内置指针定位到后半部分的第一个字节。
- 命令代码：01h。

3．Read ID

- 命令功能：表示读取 NandFlash 芯片的 ID 号。
- 命令代码：90h。

### 4．Reset

- ❑ 命令功能：表示重新启动 NandFlash 芯片。
- ❑ 命令代码：FFh。

### 5．Page Program

- ❑ 命令功能：表示对页进行编程，用于对 NandFlash 的写操作。
- ❑ 命令代码：首先写入 00h（A 区）/01h（B 区）/05h（C 区），该代码表示目标区；再写入 80h 开始编程模式，即写入模式；接着写入地址和数据；最后写入 10h 表示编程结束。

### 6．Block Erase

- ❑ 命令功能：表示对块擦除操作。
- ❑ 命令代码：首先写入 60h 进入擦写模式；再输入块地址，即将要擦除的块；接着写入 D0h 表示擦写结束。

### 7．Read Status

- ❑ 命令功能：表示读取内部状态寄存器值的命令。
- ❑ 命令代码：70h。

## 12.1.2　NandFlash 控制器

对于 2440 的 NandFlash 控制器中，寄存器有以下 12 种，与 2410 相比寄存器的设置有些变换，具体寄存器中每个 bit 的设置可以参考 2400 文档。在 S32440 芯片手册文档的第 6 章专门介绍 NandFlash 控制器时，对以下每个寄存器的设置作了详细说明。

- ❑ 配置寄存器（NFCONF）；
- ❑ 控制寄存器（NFCONT）；
- ❑ 命令寄存器（NFACMD）；
- ❑ 地址寄存器（NFADDR）；
- ❑ 数据寄存器（NFDATA）；
- ❑ 状态寄存器（NFSTAT）；
- ❑ 主数据区域 ECC 寄存器（NFMECCD0/1）；
- ❑ 空闲区域 ECC 寄存器（NFSECCD）；
- ❑ ECC0/1 状态寄存器（NFESTAT0/1）；
- ❑ 主数据区域 ECC 状态寄存器（NFMECC）；
- ❑ 空闲区域 ECC 状态寄存器（NFSECC）；
- ❑ 块地址寄存器（NFSBLK&NFEBLK）。

了解了 NandFlash 的命令和寄存器后，接下来将会介绍 NandFlash 驱动，然后介绍如何修改内核驱动使之适合 2440。在对寄存器操作时，如果有不清楚的地方可以参考 2440 文档查看对应寄存器各个位的设置情况。

# 12.2　NandFlash 驱动介绍

在 Linux 内核中已经提供了 NandFlash 驱动，驱动的声明在内核的 include/linux/mtd/nand.h 文件中，在 include/linux/mtd/nand_ecc.h 中还声明了 ECC 算法。驱动的相关实现部分主要在对应的 nand_base.c、nand_bbt.c 和 nand_ecc.c 中。下面介绍 NandFlash 驱动的主要部分。

## 12.2.1　Nand 芯片结构

结构体 nand_chip 中声明了 Nand 芯片的各种读写接口函数、buffer 操作函数、对芯片状态检查、对坏块检查和标记、芯片的属性等，下面为该结构体的定义。

```
struct nand chip {
    void  __iomem   *IO_ADDR_R;                         /*读地址*/
    void  __iomem   *IO_ADDR_W;                         /*写地址*/
    /*对字节的操作函数声明*/
    uint8_t (*read_byte)(struct mtd_info *mtd);     /*读一个字节*/
    u16     (*read_word)(struct mtd_info *mtd);     /*写一个字*/
    /*buffer 操作*/
    void    (*write_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
    void    (*read_buf)(struct mtd_info *mtd, uint8_t *buf, int len);
    int     (*verify_buf)(struct mtd_info *mtd, const uint8_t *buf, int
    len);
    /*选择一个芯片的操作*/
    void    (*select_chip)(struct mtd_info *mtd, int chip);
    /*坏块检查操作*/
    int     (*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip);
    /*坏块标记操作*/
    int     (*block_markbad)(struct mtd_info *mtd, loff_t ofs);
    /*命令控制操作*/
    void    (*cmd_ctrl)(struct mtd_info *mtd, int dat, unsigned int ctrl);
    /*设备准备操作*/
    int     (*dev_ready)(struct mtd_info *mtd);
    /*发送命令操作*/
    void    (*cmdfunc)(struct mtd_info *mtd, unsigned command, int column,
    int page_addr);
    /*等待命令完成操作*/
    int     (*waitfunc)(struct mtd_info *mtd, struct nand_chip *this);
    /*擦除操作*/
    void    (*erase_cmd)(struct mtd_info *mtd, int page);
    /*检查坏块表*/
    int     (*scan_bbt)(struct mtd_info *mtd);
    /*进行附加错误状态检查操作*/
    int     (*errstat)(struct mtd_info *mtd, struct nand_chip *this, int
            state, int status, int page);
    /*按页进行写操作*/
    int     (*write_page)(struct mtd_info *mtd, struct nand_chip *chip,
```

```
                    const uint8_t *buf, int page, int cached, int raw);
    int      chip_delay;                  /*芯片延迟*/
    unsigned int    options;              /*芯片专有选项*/
    int      page_shift;                  /*页右移的位数，即 column 地址位数*/
    int      phys_erase_shift;            /*物理擦写块的地址位数*/
    int      bbt_erase_shift;             /*坏块表入口的地址位数*/
    int      chip_shift;                  /*该芯片的地址位数*/
    int      numchips;                    /*芯片个数*/
    uint64_t    chipsize;                 /*多个芯片组中，一个芯片的大小*/
    int      pagemask;    /*每个芯片页数的屏蔽字，通过它取出每个芯片包含多少个页*/
    int      pagebuf;                     /*在页缓冲区中的页号*/
    int      subpagesize;                 /*拥有的子页大小*/
    uint8_t     cellinfo;                 /*MLC 多芯片数据*/
    int      badblockpos;                 /*坏块标记位置*/
    nand_state_t    state;                /*芯片状态*/
    uint8_t         *oob_poi;             /*缓冲区位置*/
    struct nand_hw_control  *controller;          /*硬件控制器结构体指针*/
    struct nand_ecclayout   *ecclayout;           /*默认的 ecc 设置方案*/
    struct nand_ecc_ctrl ecc;                     /*ecc 控制结构体*/
    struct nand_buffers *buffers;                 /*用于读写的缓冲区结构体*/
    struct nand_hw_control hwcontrol;             /*专用平台硬件控制结构体*/
    struct mtd_oob_ops ops;                       /*oob 操作数*/
    uint8_t     *bbt;                             /*坏块表*/
    struct nand_bbt_descr   *bbt_td;              /*坏块表描述*/
    struct nand_bbt_descr   *bbt_md;              /*坏块表映像描述*/
    struct nand_bbt_descr   *badblock_pattern;    /*坏块检测模板*/
    void        *priv;                            /*私有数据结构*/
};
```

## 12.2.2　NandFlash 驱动分析

在分析驱动时，是从普通的 Nand 驱动开始分析，到识别平台信息，然后到具体平台接口函数调用，再到对芯片寄存器读写的过程。首先看 Nand 通用驱动文件，为目录 drivers/mtd/nand 下的 plat_nand.c。

### 1．探针函数plat_nand_probe()

当系统检查到 Nand 设备时就会调用该函数，在 plat_nand_probe()函数中将参数为 platform_device 结构体的 pdev 数据 dev.platform_data，赋给了结构体 platform_nand_data，然后通过函数 platform_set_drvdata()把信息保存在 driver_data 字段中。

```
static int __init plat_nand_probe(struct platform_device *pdev)
{
    /*将参数为 platform_device 结构体的 pdev 数据 dev.platform_data，赋给了结构体
    platform_nand_data*/
    struct platform_nand_data *pdata = pdev->dev.platform_data;
    struct plat_nand_data *data;
    int res = 0;
    /*将一个 I/O 地址空间映射到内核的虚拟地址空间上，便于访问*/
```

```
    data->io_base = ioremap(pdev->resource[0].start,
                    pdev->resource[0].end - pdev->resource[0].start + 1);
    if (data->io_base == NULL) {
        dev_err(&pdev->dev, "ioremap failed\n");
        kfree(data);
        return -EIO;
    }
    /*对结构体 plat_nand_data 的各种数据、状态、命令、地址进行初始化*/
    data->chip.priv = &data;
    data->mtd.priv = &data->chip;
    data->mtd.owner = THIS_MODULE;
    data->mtd.name = dev_name(&pdev->dev);
    data->chip.IO_ADDR_R = data->io_base;
    data->chip.IO_ADDR_W = data->io_base;
    data->chip.cmd_ctrl = pdata->ctrl.cmd_ctrl;
    data->chip.dev_ready = pdata->ctrl.dev_ready;
    data->chip.select_chip = pdata->ctrl.select_chip;
    data->chip.chip_delay = pdata->chip.chip_delay;
    data->chip.options |= pdata->chip.options;
    data->chip.ecc.hwctl = pdata->ctrl.hwcontrol;
    data->chip.ecc.layout = pdata->chip.ecclayout;
    data->chip.ecc.mode = NAND_ECC_SOFT;
    /*使用函数 platform_set_drvdata()将信息保存在设备的 driver_data 字段中*/
    platform_set_drvdata(pdev, data);
    /*扫描是否存在 mtd 设备*/
    if (nand_scan(&data->mtd, 1)) {
        res = -ENXIO;
        goto out;
    }
#ifdef CONFIG_MTD_PARTITIONS
    if (pdata->chip.part_probe_types) {
        res = parse_mtd_partitions(&data->mtd,
                    pdata->chip.part_probe_types,
                    &data->parts, 0);
        if (res > 0) {
            add_mtd_partitions(&data->mtd, data->parts, res);
            return 0;
        }
    }
    if (pdata->chip.partitions) {
        data->parts = pdata->chip.partitions;
        res = add_mtd_partitions(&data->mtd, data->parts,
            pdata->chip.nr_partitions);
    } else
#endif
    /*添加 mtd 设备*/
    res = add_mtd_device(&data->mtd);
    if (!res)
        return res;
    /*释放 nand 设备占用的资源*/
    nand_release(&data->mtd);
out:
    /*资源释放部分：释放设备占用资源、释放 I/O 映射到内存的空间、释放 data 占用空间*/
    platform_set_drvdata(pdev, NULL);
    iounmap(data->io_base);
    kfree(data);
    return res;
}
```

### 2. platfrom_nand_data结构体

platfrom_nand_data 结构体在 include/linux/mtd 中进行的定义。该结构体包括特定平台的数据信息，在该结构体中包括两项内容，即特定芯片结构体和设备控制器结构体。下面为这三个结构体的定义。

```
/*platform_nand_data 结构体的定义*/
struct platform_nand_data {
    struct platform_nand_chip   chip;
    struct platform_nand_ctrl   ctrl;
};
/*platform_nand_chip 结构体定义特定芯片的属性*/
struct platform_nand_chip {
    int          nr_chips;
    int          chip_offset;
    int          nr_partitions;
    struct mtd_partition  *partitions;
    struct nand_ecclayout  *ecclayout;
    int          chip_delay;
    unsigned int        options;
    const char      **part_probe_types;
    void            *priv;
};
/*platform_nand_ctrl 结构体中定义了对芯片控制的操作函数*/
struct platform_nand_ctrl {
    void         (*hwcontrol)(struct mtd_info *mtd, int cmd);
    int          (*dev_ready)(struct mtd_info *mtd);
    void         (*select_chip)(struct mtd_info *mtd, int chip);
    void         (*cmd_ctrl)(struct mtd_info *mtd, int dat,
                    unsigned int ctrl);
    void         *priv;
};
```

### 3. 驱动初始化s3c24xx_nand_init()

加载该驱动后，就会注册 s3c2440_nand_driver 驱动，因为 Mini2440 的 CPU 类型为 S3C2440，因此在函数调用 s3c2410_nand_init 时，注册支持 Mini2440 的 NandFlash 驱动。

```
module_init(s3c2410_nand_init);
static int __init s3c2410_nand_init(void)
{
    printk("S3C24XX NAND Driver, (c) 2004 Simtec Electronics\n");
    platform_driver_register(&s3c2412_nand_driver);
    /*注册 s2c2440 _nand_driver*/
    platform_driver_register(&s3c2440_nand_driver);
    return platform_driver_register(&s3c2410_nand_driver);
}
/*s3c2440_nand_driver 结构体的定义如下*/
static struct platform_driver s3c2440_nand_driver = {
    .probe      = s3c2440_nand_probe,
    .remove     = s3c2410_nand_remove,
    .suspend = s3c24xx_nand_suspend,
    .resume     = s3c24xx_nand_resume,
    .driver     = {
        .name   = "s3c2440-nand",
        .owner  = THIS_MODULE,
```

```
    },
};
```

### 4．探针函数s3c2440_nand_probe()

因为调用函数 platform_driver_register（&s3c2440_nand_driver）后，调用探针函数时，函数 s3c2440_nand_probe 就会被调用。

```
static int s3c2440_nand_probe(struct platform_device *dev)
{
    /*第 2 个参数指定了 CPU 的类型为 TYPE_S3C2440*/
    return s3c24xx_nand_probe(dev, TYPE_S3C2440);
}
/*函数 s3c24xx_nand_probe()的定义*/
static  int  s3c24xx_nand_probe(struct  platform_device  *pdev,  enum
s3c_cpu_type cpu_type)
{
    /*分配空间并进行初始化*/
    info = kmalloc(sizeof(*info), GFP_KERNEL);
    memset(info, 0, sizeof(*info));
    /*将 info 保存在 driver_data 字段中*/
    platform_set_drvdata(pdev, info);
    spin_lock_init(&info->controller.lock);
    /*初始化队列*/
    init_waitqueue_head(&info->controller.wq);
    /*获得时钟资源并开启*/
    info->clk = clk_get(&pdev->dev, "nand");
    clk_enable(info->clk);
    /*分配和映射资源*/
    res  = pdev->resource;
    size = res->end - res->start + 1;
    /*为 NandFlash 寄存器区申请 I/O 内存地址空间区，并通过 ioremap()把它映射到虚拟地
    址空间*/
    info->area = request_mem_region(res->start, size, pdev->name);
    info->device       = &pdev->dev;
    info->platform     = plat;
    info->regs         = ioremap(res->start, size);
    info->cpu_type     = cpu_type;
    /*初始化 NandFlash 控制器*/
    s3c2410_nand_inithw(info);
    /*为 mtd 设备分配设备信息的存储空间*/
    size = nr_sets * sizeof(*info->mtds);
    kmalloc(size, GFP_KERNEL);
    memset(info->mtds, 0, size);
    /*初始化所有可能的芯片*/
    nmtd = info->mtds;
    for (setno = 0; setno < nr_sets; setno++, nmtd++) {
        s3c2410_nand_init_chip(info, nmtd, sets);
        nmtd->scan_res = nand_scan_ident(&nmtd->mtd,
                         (sets) ? sets->nr_chips : 1);
        if (nmtd->scan_res == 0) {
            s3c2410_nand_update_chip(info, nmtd);
```

```
            nand_scan_tail(&nmtd->mtd);
            s3c2410_nand_add_partition(info, nmtd, sets);
        }
        if (sets != NULL)
            sets++;
    }
    /*CPU 频率驱动*/
    s3c2410_nand_cpufreq_register(info);
    if (allow_clk_stop(info)) {
        dev_info(&pdev->dev, "clock idle support enabled\n");
        clk_disable(info->clk);
    }
    return 0;
}
```

### 5．支持的芯片类型s3c_cpu_type

在 s3c2410.c 中还定义了支持的芯片类型，枚举类型 s3c_cpu_typ 中定义了支持的芯片类型。

```
enum s3c_cpu_type {
    TYPE_S3C2410,
    TYPE_S3C2412,
    TYPE_S3C2440,
};
```

### 6．控制器初始化s3c2410_nand_inithw()

在函数 s3c24xx_nand_probe()中调用函数 s3c2410_nand_inithw()对 NandFlash 控制器进行初始化。在对硬件初始化函数 s3c2410_nand_inithw()中就涉及了对寄存器的访问。

```
static int s3c2410_nand_inithw(struct s3c2410_nand_info *info)
{
    int ret;
    ret = s3c2410_nand_setrate(info);
    if (ret < 0)
        return ret;
    /*根据 CPU 型号写不同的配置*/
    switch (info->cpu_type) {
    case TYPE_S3C2410:
    default:
        break;
    case TYPE_S3C2440:
    case TYPE_S3C2412:
        /*以四字节形式向配置寄存器中写内容。S3C2440_NFCONT_ENABLE 的值为 1，即向
        置寄存器 NFCONT 中写 1，通过查看芯片资料，对应 bit0 为 1 时表示使能 NandFlash
        控制，其初始状态为 0*/
        writel(S3C2440_NFCONT_ENABLE, info->regs + S3C2440_NFCONT);
    }
    return 0;
}
```

对于 Nand 驱动的其他函数和操作也可以根据相同的方法进行分析，在对驱动源码进行分析和理解的基础上，下面将介绍如何针对具体芯片进行移植。

# 12.3　NandFlash 驱动移植

移植 NandFlash 驱动时只要对内核代码作少量修改即可，修改的内容主要包括对 NandFlash 类型的支持、NandFlash 分区和 NandFlash 注册。

## 12.3.1　内核的修改

### 1．NandFlash类型的支持

文件 nand_ids.c 中的数组 nand_flash_ids 中定义了内核支持的各种不同容量的 NandFlash 芯片，包括分页大小、容量大小等信息。

```
struct nand_flash_dev nand_flash_ids[] = {
    /*512 Megabit*/
    {"NAND 64MiB 1,8V 8-bit",   0xA2, 0,  64, 0, LP_OPTIONS},
    {"NAND 64MiB 3,3V 8-bit",   0xF2, 0,  64, 0, LP_OPTIONS},
    {"NAND 64MiB 1,8V 16-bit",  0xB2, 0,  64, 0, LP_OPTIONS16},
    {"NAND 64MiB 3,3V 16-bit",  0xC2, 0,  64, 0, LP_OPTIONS16},

    /*1 Gigabit*/
    {"NAND 128MiB 1,8V 8-bit",  0xA1, 0, 128, 0, LP_OPTIONS},
    {"NAND 128MiB 3,3V 8-bit",  0xF1, 0, 128, 0, LP_OPTIONS},
    {"NAND 128MiB 1,8V 16-bit",0xB1, 0, 128, 0, LP_OPTIONS16},
    {"NAND 128MiB 3,3V 16-bit",0xC1, 0, 128, 0, LP_OPTIONS16},

    /*2 Gigabit*/
    {"NAND 256MiB 1,8V 8-bit",  0xAA, 0, 256, 0, LP_OPTIONS},
    {"NAND 256MiB 3,3V 8-bit",  0xDA, 0, 256, 0, LP_OPTIONS},
    {"NAND 256MiB 1,8V 16-bit",0xBA, 0, 256, 0, LP_OPTIONS16},
    {"NAND 256MiB 3,3V 16-bit",0xCA, 0, 256, 0, LP_OPTIONS16},

    /*4 Gigabit*/
    {"NAND 512MiB 1,8V 8-bit",  0xAC, 0, 512, 0, LP_OPTIONS},
    {"NAND 512MiB 3,3V 8-bit",  0xDC, 0, 512, 0, LP_OPTIONS},
    {"NAND 512MiB 1,8V 16-bit",0xBC, 0, 512, 0, LP_OPTIONS16},
    {"NAND 512MiB 3,3V 16-bit",0xCC, 0, 512, 0, LP_OPTIONS16},

    /*8 Gigabit*/
    {"NAND 1GiB 1,8V 8-bit",    0xA3, 0, 1024, 0, LP_OPTIONS},
    {"NAND 1GiB 3,3V 8-bit",    0xD3, 0, 1024, 0, LP_OPTIONS},
    {"NAND 1GiB 1,8V 16-bit",   0xB3, 0, 1024, 0, LP_OPTIONS16},
    {"NAND 1GiB 3,3V 16-bit",   0xC3, 0, 1024, 0, LP_OPTIONS16},

    /*16 Gigabit*/
    {"NAND 2GiB 1,8V 8-bit",    0xA5, 0, 2048, 0, LP_OPTIONS},
    {"NAND 2GiB 3,3V 8-bit",    0xD5, 0, 2048, 0, LP_OPTIONS},
    {"NAND 2GiB 1,8V 16-bit",   0xB5, 0, 2048, 0, LP_OPTIONS16},
    {"NAND 2GiB 3,3V 16-bit",   0xC5, 0, 2048, 0, LP_OPTIONS16},
};
```

### 2. NandFlash分区

在文件 mach-mini2440.c 中修改 NandFlash 分区表，文件 mach-mini2440.c 可以参考 mach-smdk2440.c 进行修改。

```
static struct mtd_partition mini2440_default_nand_part[] = {
    [0] = {
            .name   = "supervivi",          /*bootloader 所在分区*/
            .size   = 0x00030000,
            .offset = 0,
    },
    [1] = {
            .name   = "Kernel",             /*内核所在分区*/
            .offset  = 0x00050000,
            .size   = 0x00200000,
    },
    [2] = {
            .name   = "root",               /*文件系统所在分区*/
            .offset  = 0x00250000,
            .size   = 0x03dac000,
    }
};
```

### 3. NandFlash注册

将 NandFlash 设备注册到系统中，在__initdata 中添加 NandFlash 设备。

```
/*开发板上所有 NandFlash 的设置表，mini2440 上只有一块 NandFlash，如果有多块在后面
继续添加*/
static struct s3c2410 nand set mini2440 nand sets[] = {
    [0] = {
            .name       = "NAND",
            .nr_chips = 1,
            .nr_partitions  = ARRAY_SIZE(mini2440_default_nand_part),
            .partitions = mini2440_default_nand_part,
    },
};
/*NandFlash 信息*/
static struct s3c2410 platform nand mini2440 nand info = {
    .tacls      = 20,
    .twrph0     = 60,
    .twrph1     = 20,
    .nr_sets  = ARRAY_SIZE(mini2440_nand_sets),
    .sets  = mini2440_nand_sets,
};
/*将 NandFlash 设备注册到系统中*/
static struct platform_device *mini2440_devices[] __initdata = {
    &s3c device usb,
    &s3c device rtc,
    &s3c device lcd,
    &s3c_device_wdt,
    &s3c_device_i2c0,
    &s3c_device_iis,
    &s3c device dm9k,
    &net device cs8900,
    &s3c24xx uda134x,
    &s3c_device_sdi,
    &s3c device nand,
};
```

### 12.3.2　内核的配置和编译

在编译内核时，配置对 MTD 的支持选项 Memory Technology Device (MTD) support。进入该配置窗口后，选择 MTD 分区支持 MTD partitioning support，配置如图 12.1 所示。

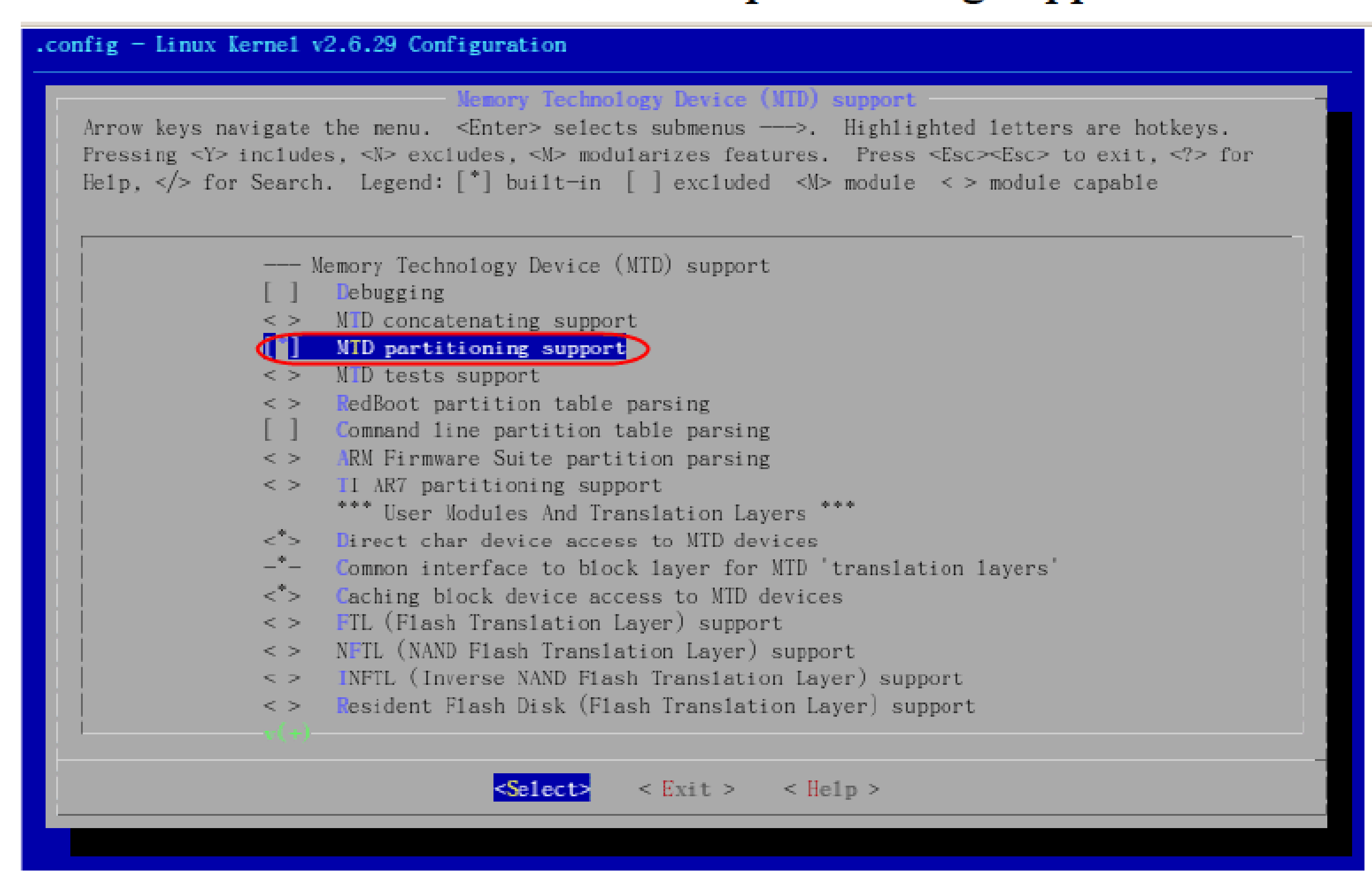



图 12.1　配置 MTD 分区支持

进入 NAND Device Support 配置窗口选择对具体芯片类型的支持，配置如图 12.2 所示。

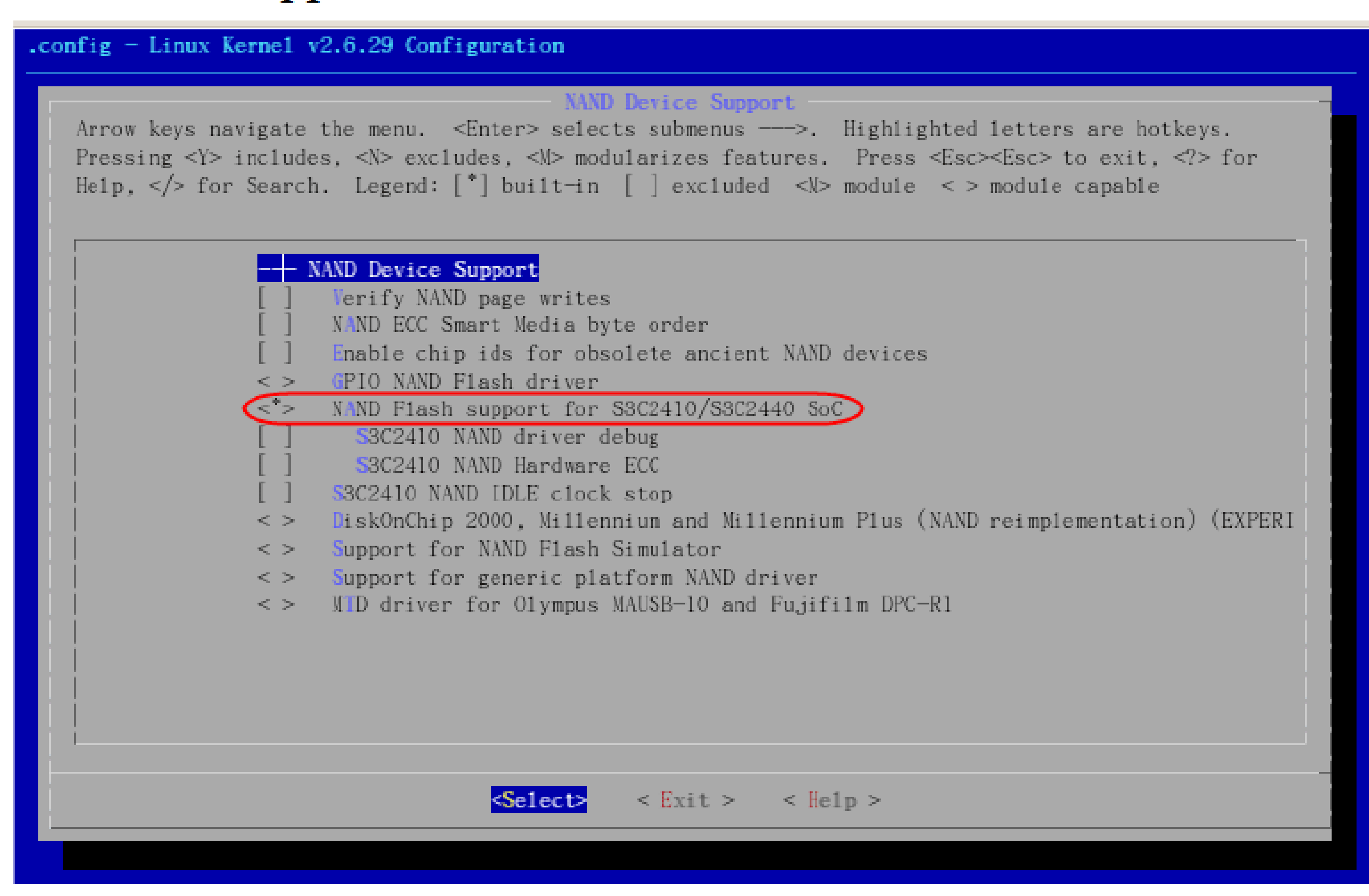



图 12.2　配置对 S3C2440 的支持

## 12.4　小　　结

NandFlash 驱动移植是比较简单的内容，对于其烧写内核部分省略了，读者可以参考前面的章节关于介绍内核编译、移植和烧写的过程。本章主要介绍分析 NandFlash 驱动的实现过程，达到让读者能根据不同型号的 NandFlash 修改对应驱动文件和移植的目的。

# 第 4 篇　系统移植高级篇

# 第 13 章　MiniGUI 与移植

MiniGUI 是根据嵌入式系统应用特点量身定做的图形支持系统。MiniGUI 支持多种操作系统，如 uClinux、VxWorks、eCos、uC/OS-II、ThreadX、Nucleus、OSE 等。它也可以运行在 Win32 平台。本章主要介绍 MiniGUI 在上位机中的安装、开发和调试方法，以及移植到 ARM 平台的过程。

## 13.1　MiniGUI 在上位机中的安装

MiniGUI 在移植到开发板之前，必须先在上位机中调试通过，然后交叉编译，最后移植到开发板上。下面将以 MiniGUI-1.12.10 在 Fedora Core release 6 版本上的安装过程为例，介绍 MiniGUI 的安装和编译过程。

### 13.1.1　安装需要的安装文件

在表 13.1 中列出了需要的安装文件和 RPM 包。如果需要对各个文件进行更详细的了解，可以查看 MiniGUI 的手册文件。

表 13.1　安装文件和RPM包列表

| 安 装 文 件 | 说　　明 |
|---|---|
| libminigui-1.6.10.tar.gz | MiniGUI 的函数库源代码 |
| jpegsrc.v6b.tar.gz | MiniGUI 依赖函数库，用来支持 JPEG 图片 |
| libpng_src.gz | MiniGUI 依赖函数库，用来支持 PNG 图片 |
| mde-1.6.10.tar.gz | MiniGUI 的综合演示程序包，是一些复杂的例子 |
| mg-samples-1.6.10.tar.gz | 《MiniGUI 编程指南》的示例程序 |
| minigui-res-1.6.10.tar.gz | MiniGUI 所使用的资源文件，包括字体、图标、位图和鼠标光标等 |
| qvfb-1.1.tar.gz | MiniGUI 的图形引擎 |
| samples-1.6.10.tar.gz | MiniGUI 丰富的例子程序，包括 mGPoint、mGPaint、eHomeSample、IndustrialSample、MedicalSample、STBSample 等 |
| RPM 包 | |
| qt-devel-3.3.6-13.i386.rpm | 安装后得到补充的 QT 库文件和头文件 |

- 安装文件的下载地址为：http://sourceforge.net/projects/minigui/files/minigui/GPL-V1.6.10/。
- RPM 文件的下载地址为：http://ftp.isu.edu.tw/pub/Linux/Fedora/linux/extras/6/i386/?page=84

## 13.1.2　MiniGUI 的运行模式

MiniGUI 可以配置成 4 种模式中的一种：多进程的 MiniGUI-Processes、MiniGUI-Lite 运行模式、多线程的 MiniGUI-Threads 运行模式和非多线程的 MiniGUI-Standalone 运行模式。MiniGUI 不同的版本包括不同的 MiniGUI 运行模式。

- MiniGUI-Processes 运行模式：MiniGUI-Processes 上的每个程序是单独的进程，每个进程也可以建立多个窗口，并实现了多进程串口系统，来自不同进程的窗口可以在同一桌面上协调存在。
- MiniGUI-Lite 运行模式：MiniGUI-Lite 实现了不同窗口之间的切换，但没能够解决进程间窗口层叠的问题，无法同时管理来自不同进程间的窗口。在 MiniGUI 2.0 后被 MiniGUI-Processes 运行模式取代。
- MiniGUI-Threads 运行模式：MiniGUI-Threads 可以在不同的线程中建立多个窗口，所有的窗口在同一个进程中。
- MiniGUI-Standalone 运行模式：MiniGUI-Standalone 不需要多进程和多线程的支持，适合功能单一的应用场合。

MiniGUI 安装时 MiniGUI 运行模式的配置方法，如表 13.2 所示。

表 13.2　MiniGUI运行模式的配置方法

| Configure 脚本选项 | 宏 | 备　注 | 默认 |
| --- | --- | --- | --- |
| 不指定 | _MGRM_THREADS | MiniGUI-Threads 运行模式 | |
| procs | _MGRM_PROCESSES<br>_LITE_VERSION | MiniGUI 运行模式，仅用于 Linux/uClinux 操作系统 | 关闭 |
| standalone | _MGRM_STANDALONE<br>_LITE_VERSION<br>_STAND_ALONE | MiniGUI-Standalone 运行模式，仅用于 Linux/uClinux 操作系统 | 关闭 |

目前嵌入式的芯片功能比较强大，各种嵌入式应用场合也越来越复杂，下面的运行模式配置将配置成 MiniGUI-Threads 运行模式，即直接执行./configure 即可。

## 13.1.3　编译并安装 MiniGUI

编译 MiniGUI 采用的内核版本为 Linux-2.6.18 的 Fedora Core release 6 版本；编译器的版本为 gcc-4.1.1；MiniGUI 的版本为 1.6.10。读者可以使用 uname –a 查看本机的内核版本，cat /etc/issue 查看发行的 Linux 版本，使用 gcc –v 查看本机默认的编译器版本。

```
#uname -a                  //查看内核版本
#cat/etc/issue             //查看 Linux 发行版本
#gcc -v                    //查看 GCC 版本
```

### 1．安装MiniGUI的函数库（libminigui-1.6.10.tar.gz）

（1）首先建立存放安装源文件目录，将需要安装的文件放在该目录下。

```
#mkdir/usr/local/minigui_setup          //建立存放安装文件目录
```

（2）完成安装文件的复制后，解压安装文件，解压方法如下：

```
#cd/usr/local/minigui setup
#tar -zxvf libminigui-1.6.10.tar.gz    //解压 libminigui-1.6.10.tar.gz
```

（3）进入解压后的目录进行安装。

```
#cd libminigui-1.6.10
#./configure           //安装默认的配置，将其配置成 MiniGUI-Threads 运行模式
#make                  //执行编译
#make install          //安装，默认安装在/usr/local 目录下
```

通过 ls 命令可以查看安装的结果。

```
#ls/usr/local/lib                         //查看安装的库文件
libmgext-1.6.so.10      libmgext.so             libminigui.la
libvcongui.a
libmgext-1.6.so.10.0.0  libminigui-1.6.so.10    libminigui.so
libvcongui.la
libmgext.a              libminigui-1.6.so.10.0.0    libvcongui-1.6.so.10
libvcongui.so
libmgext.la             libminigui.a            libvcongui-1.6.so.10.0.0
#ls/usr/local/include/minigui/            //查看安装的头文件
colordlg.h endianrw.h  mgext.h     ose_semaphore.h threadx_pthread.h
vxworks_semaphore.h
colorspace.h  ext      minigui.h   own_malloc.h    threadx_semaphore.h
win32 dirent.h
common.h filedlg.h     mywindows.h own stdio.h     ucos2 pthread.h
win32_pthread.h
control.h              fixedmath.h newfiledlg.h    psos_pthread.h
ucos2_semaphore.h   win32_sched.h
ctrl  gdi.h      nucleus_pthread.h  psos_semaphore.h    vcongui.h
win32 semaphore.h
dti.c    mgconfig.h    nucleus semaphore.h    skin.h  vxworks pthread.h
window.h
```

### 2．安装MiniGUI的资源文件（minigui-res-1.6.10.tar.gz）

```
# tar -zxvf minigui-res-1.6.10.tar.gz      //解压资源文件包
# cd minigui-res-1.6.10                    //进入解压后的资源文件目录
# make install                             //执行安装
```

**注意：** 读者在安装的过程中应该以超级用户的身份进行安装，这里的安装过程都是以超级用户的身份执行的。

### 3．添加共享库的搜索路径

在/etc/ld.so.conf 文件中包含了默认的共享搜索路径。在该文件的后面添加刚才安装的 MiniGUI 的库文件路径。

```
#vi /etc/ld.so.conf
```

在文件 ld.so.conf 末尾添加/usr/local/lib 和/usr/lib。

```
#ldconfig                              //刷新共享库缓存
```

### 4．安装RPM（qt-devel-3.3.6-13.i386.rpm）

```
# rpm -i --force qt-devel-3.3.6-13.i386.rpm
                                       //安装支持 Qt 3.0.3 的头文件和库文件
```

**注意：** 如果不安装此 rpm 包，在安装配置 QVFB 时会提示找不到相关的库文件和头文件。checking for Qt... configure: error: Qt (>= Qt 3.0.3) (headers and libraries) not found. Please check your installation!

### 5．安装qvfb

qvfb 是 Qt 提供的一个虚拟 FrameBuffer（帧缓存）工具。这个工具是基于 Qt 开发，运行在 XWindow 上。

```
#tar -zxvf qvfb-1.1.tar.gz
#cd qvfb-1.1
./configure  \
--with-qt-includes=/usr/lib/qt-3.3/include \        //指定头文件目录
--with-qt-libraries=/usr/lib/qt-3.3/lib    \        //指定库文件目录
--with-qt-dir=/usr/lib/qt-3.3                       //指定路径
#make
#make install
```

### 6．执行qvfb测试qvfb

```
#qvfb
```

安装成功后，运行 qvfb 可以看到如图 13.1 所示的图形界面，从现在开始可以在此基础上开发和调试自己的产品了。此时的界面中不包含任何控件、按钮和图片。

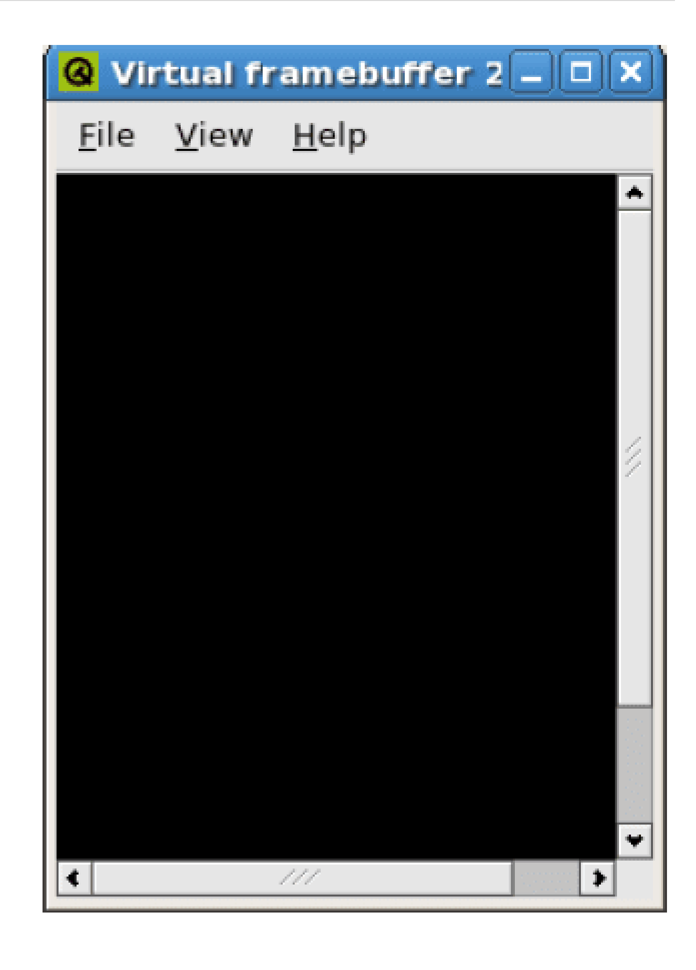



图 13.1　qvfb 测试结果

**注意：** 在有些书上和 MiniGUI 的手册上，都讲了对 MiniGUI 运行时配置文件/usr/locla/etc/MiniGUI.cfg 的设置。本机默认的配置图形引擎就是 qvfb，所以不需要进一步的配置文件/usr/locla/etc/MiniGUI.cfg。如果读者的计算机上无法正确显示，请参考本文的配置修改。下面给出本机的配置供参考。qvfb 段配置：

```
[system]
# GAL engine and default options
gal_engine=qvfb                      //指定 qvfb 作为图形引擎
defaultmode=800x600-16bpp

# IAL engine
ial_engine=qvfb
mdev=/dev/input/mice
mtype=IMPS2
```

```
[fbcon]
defaultmode=1024x768-16bpp

[qvfb]
defaultmode=640x480-16bpp
display=0
```

### 13.1.4　编译安装 MiniGUI 需要的图片支持库

安装支持 png 格式的文件 libpng_src.gz 和支持 jgeg 格式的文件 jpegsrc.v6b.tar.gz。

```
#tar zxvf jpegsrc.v6b.tar.gz
#cd jpeg-6b
#make
# mkdir /usr/local/man                  //创建 man1 目录，手册安装在此目录下
#mkdir /usr/local/man/man1              //创建 man1 目录，手册安装在此目录下
#make install
# tar zxvf libpng src.gz
# cd libpng
# make -l /usr/lib/                     //指定编译库，否则会出现找不到 libpng.a 错误
#make install
```

### 13.1.5　编译 MiniGUI 应用程序例子

解压例子程序包，通过这些例子让读者很快熟悉 MiniGUI 的编程格式，下面是这些例子的解压、编译和运行过程。

```
# tar -zxvf mg-samples-1.6.10.tar.gz
# cd mg-samples-1.6.10
# ./configure
# make
```

执行 make 后，会在 src 子目录中生成可执行文件。直接运行例子程序的时候会遇到无法初始化图形引擎的问题，错误的情况类似下面的提示：

```
NEWGAL: Video mode smaller than requested.
NEWGAL: Set video mode failure.
InitGUI: Can not initialize graphics engine!
```

其原因是默认的输出模式 defaultmode 设置问题，在配置/usr/locla/etc/MiniGUI.cfg 文件的时候，提到了 defaultmode=640x480-16bpp。重新开启另外一个终端，执行 qvfb，选择命令 File->Configure，修改其模式为 640×480，且单击 Ok 按钮改变其显示模式，如图 13.2 所示。然后再运行例子程序，效果如图 13.3 所示。或者通过命令形式指定其显示模式，然后再运行例子程序可以得到与图 13.3 同样的效果。

```
#qvfb -width 640 -height 480 -depth 16
```

通过上面的介绍，读者可以自己修改例子程序，编译运行查看效果。接下来将介绍本章的另外两个重点，MiniGUI 的常用的开发方法和 MiniGUI 程序的交叉编译和移植过程。

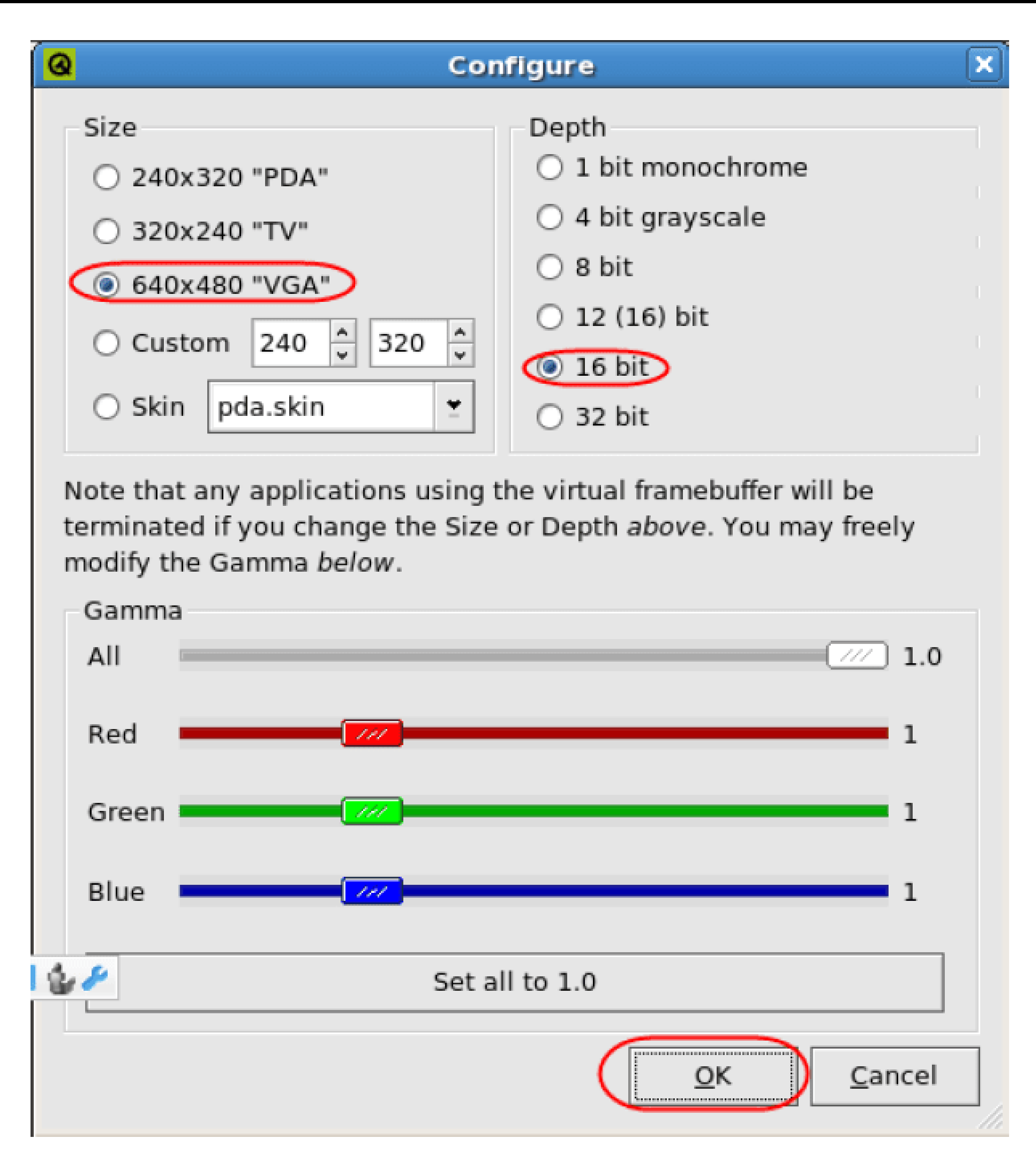



图 13.2　设置显示模式大小

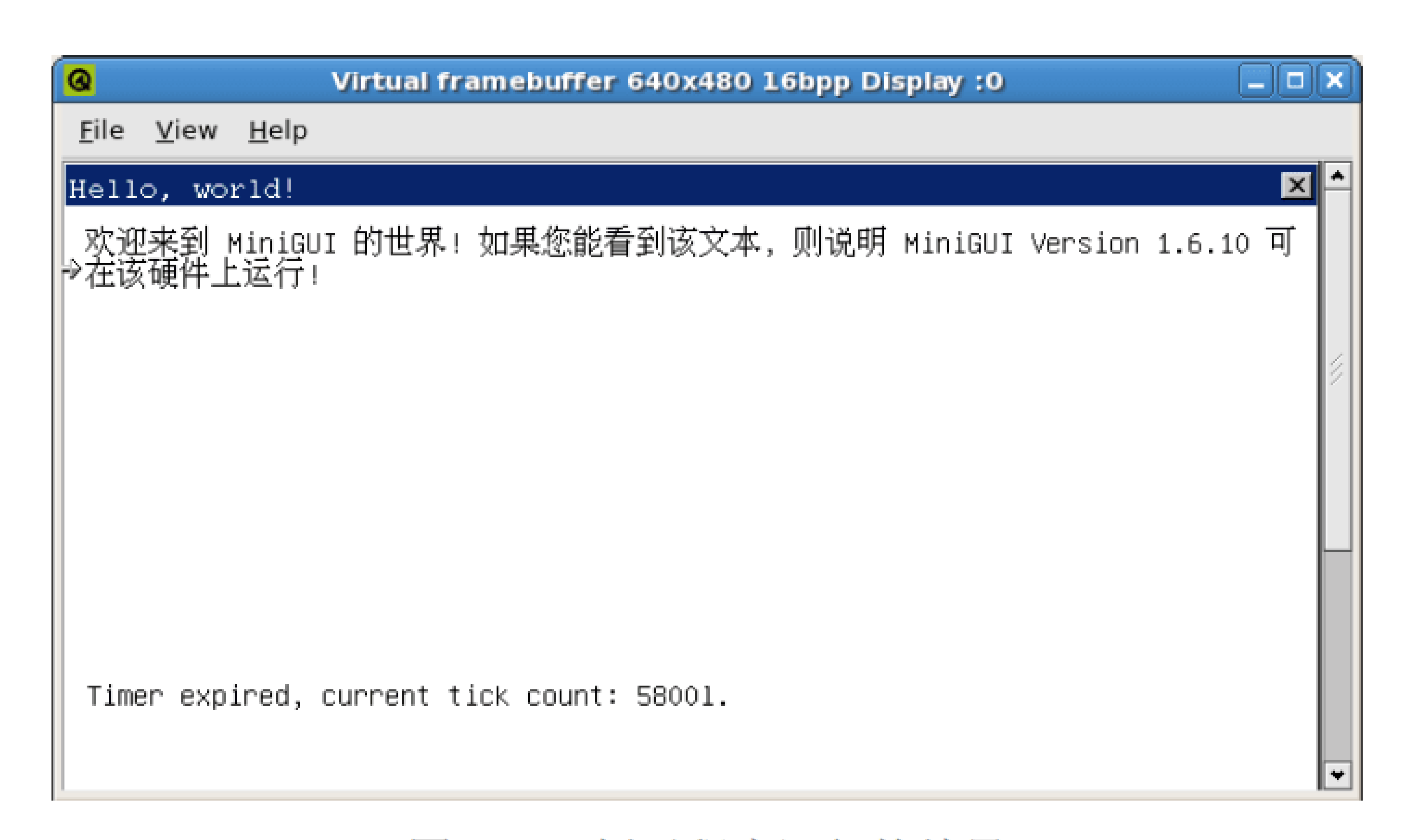



图 13.3　例子程序运行的结果

# 13.2　Eclipse 开发 MiniGUI 程序

通过 IDE 编程工具进行代码的管理和开发对于较大的工程比较方便。相比通过命令行进行编译和对代码进行修改，IDE 编译器具有绝对的优势。下面介绍如何在 Linux 中使用 Eclipse 进行开发 MiniGUI 程序。

## 13.2.1　Linux 下安装 Eclipse 介绍

在 Fedora 版本中带有 Eclipse 安装包，如图 13.4 所示。如果在安装系统的时候没有添

加 Eclipse，可以通过选择“应用程序|添加/删除软件”命令添加 Eclipse。

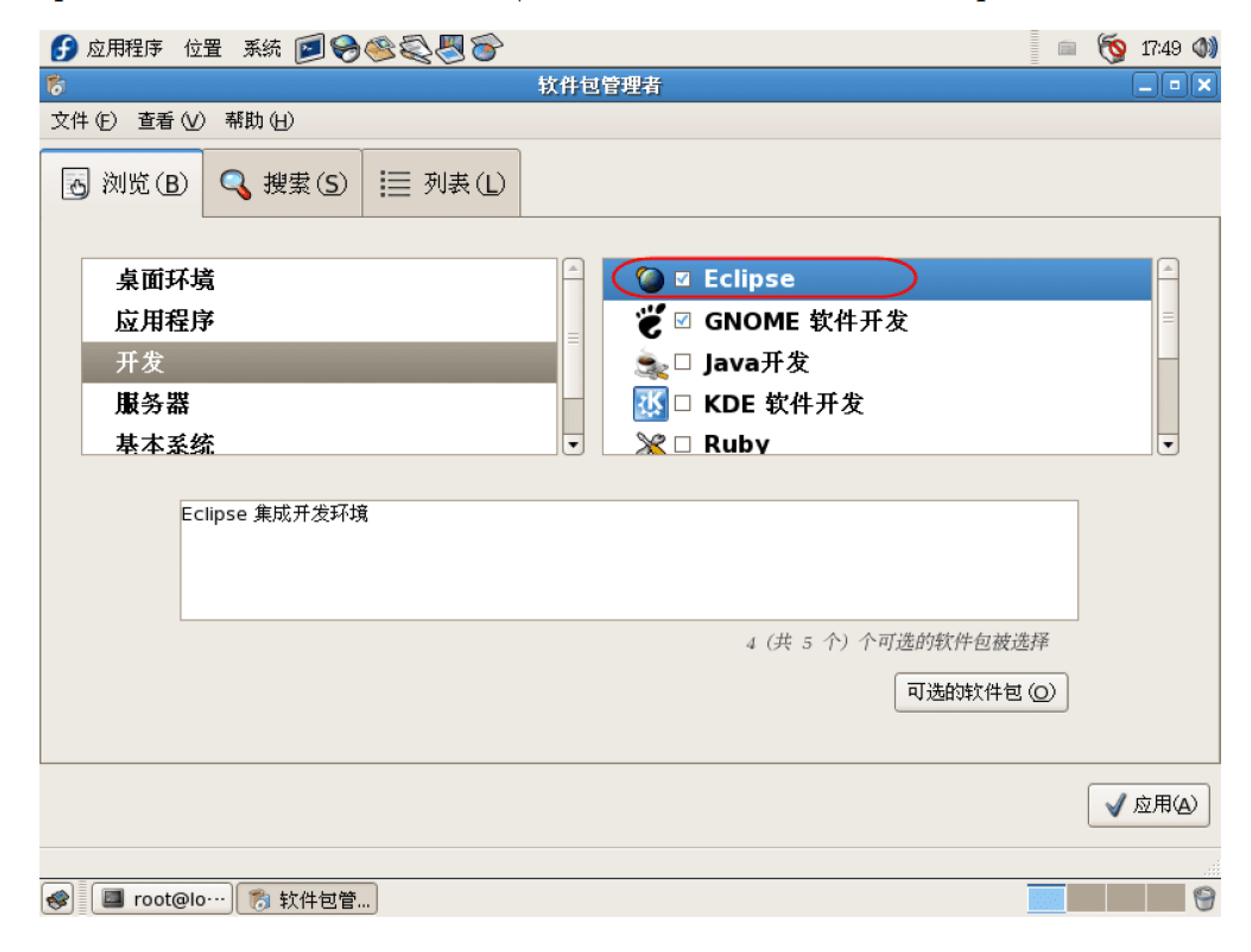



图 13.4 安装 Eclipse

如果出现“无法安装软件包”错误提示，则按照下面方法进行安装，安装过程根据不同的系统版本会有差异。本机的系统版本为 FC6，其他的系统安装方法可以进行参考，详细过程不进行贴图说明，读者可以在网上查阅相关资料。

```
#mount -t iso9660 /dev/cdrom /mnt/cdrom/        //挂载光驱
#cd /etc/yum.repos.d/                           //建立添加删除配置文件
vi cdrom.repo
[cdrom]                                         //配置文件 cdrom.repo 的内容
name=Fedora software from cdrom
baseurl=file:///mnt/cdrom
#vi /usr/lib/python2.4/site-packages/yum/yumRepo.py
                                                //修改配置文件 yumRepo.py
remote = url + '/' + relative       //找到该行，将 url 改为刚才挂载的光驱路径
remote = "/mnt/cdrom" + '/' + relative          //修改后
#cd /etc/yum.repos.d/
[root@localhost yum.repos.d]# vi fedora-core.repo
                                    //修改 enabled=1 为 enabled=0
[root@localhost yum.repos.d]# vi fedora-updates.repo
                                    //修改 enabled=1 为 enabled=0
[root@localhost yum.repos.d]# vi fedora-extras.repoe
                                    //修改 enabled=1 为 enabled=0
```

安装过程中可能需要依赖其他的 RPM 包，可以直接进入/mnt/cdrom/Fedora/RPMS 找到相应的 RPM 包进行安装。默认安装方式 Eclipse 是不支持 C/C++工程，与前面安装方式相同，进入/mnt/cdrom/Fedora/RPMS 目录下安装 RPM 文件 eclipse-cdt-3.1.1-1.fc6.i386.rpm。

安装后新建工程向导中出现 C/C++工程选项，如图 13.5 所示。

图 13.5　增加 C/C++工程选项

## 13.2.2　使用 Eclipse 编译 MiniGUI 程序

对于有过 Visual C++（简称 VC++）编程经验或者用 Eclipse 编写 Java 经验的读者来说，用 Eclipse 编写 C/C++代码具有语法错误提示功能，以使用 VI 更有优势，能方便生成 Makefile 文件，方便设计和生成文档。本节将通过 MiniGUI 的示例程序进行说明 Eclipse 编译 MiniGUI 的过程及相关的设置。

### 1. 使用Eclipse创建并运行项目

（1）解压源码，利用主目录下的 configure 文件生成 Makefile 文件，阅读 Makefile 文件查看其需要的头文件和相关的库文件。

Eclipse 可以帮助生成 Makefile，这里以 mde-1.6.10.tar.gz 中的 bomb 程序为例，介绍整个项目的导入、编译及运行过程。在设置编译配置的时候可以先通过源代码的 configure 文件生成 Makefile，然后参考 Makefile 对 Eclipse 的编译配置进行设置，或者通过读源代码和《MiniGUI 编程指南》。本文的编译器设置是通过读 Makefile 来设置。

（2）使用 File | New | Project 建立 C 项目，如图 13.6 所示。根据建立项目向导制定项目名称并完成建立项目过程，在创建工程的时候，Project Type 选择 Executable(Gnu)。在 Project 视图中右击项目，在弹出的快捷菜单中选择 Import 命令导入项目文件，导入对话框如图 13.7 所示。在导入对话框中通过单击 Browse 按钮选择文件所在的目录。

### 2. 配置编译选项

在调试运行项目前需要配置编译选项，包括编译器配置和连接器配置等参数，将

MiniGUI 的头文件目录、jpeg 支持、png 支持等头文件目录包含到编译器的头文件目录，如图 13.8 所示。为编译指定依赖的各种库，如图 13.9 所示。

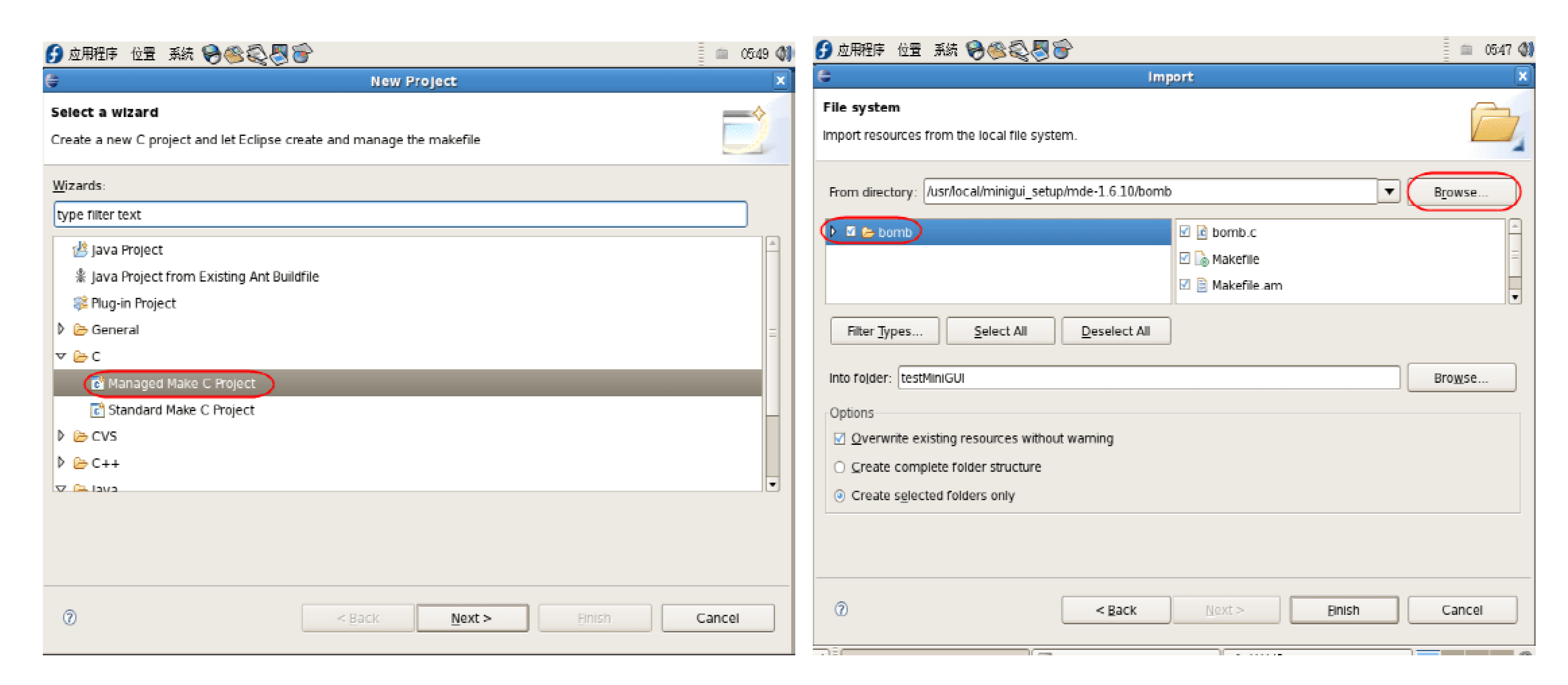



图 13.6　建立项目　　　　图 13.7　导入项目

添加的头文件路径包括：

```
/usr/local/include/minigui
/usr/local/include
/usr/include
```

添加的库包括：

```
minigui
pthread
png
jpeg
m
```

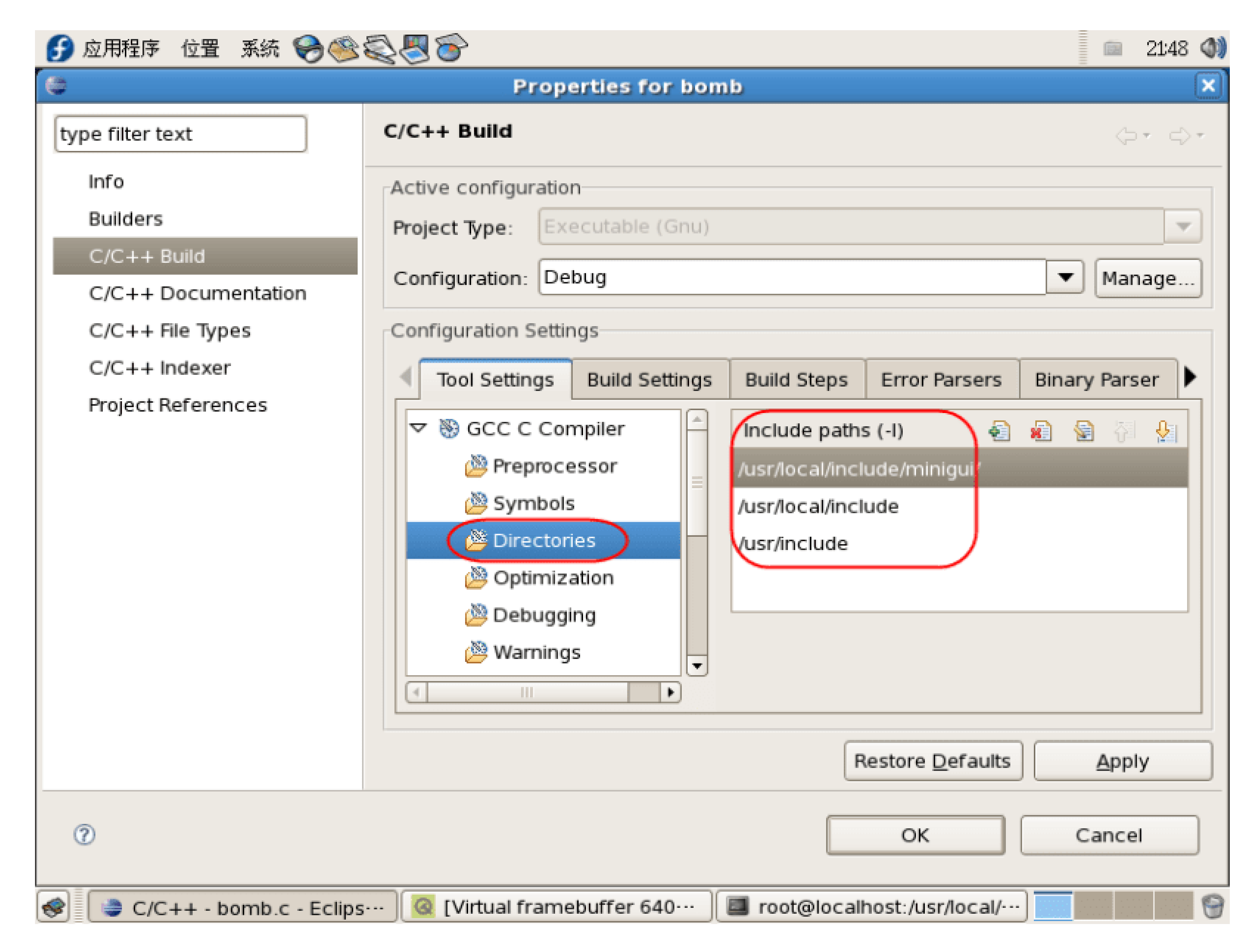



图 13.8　指定编译头文件目录

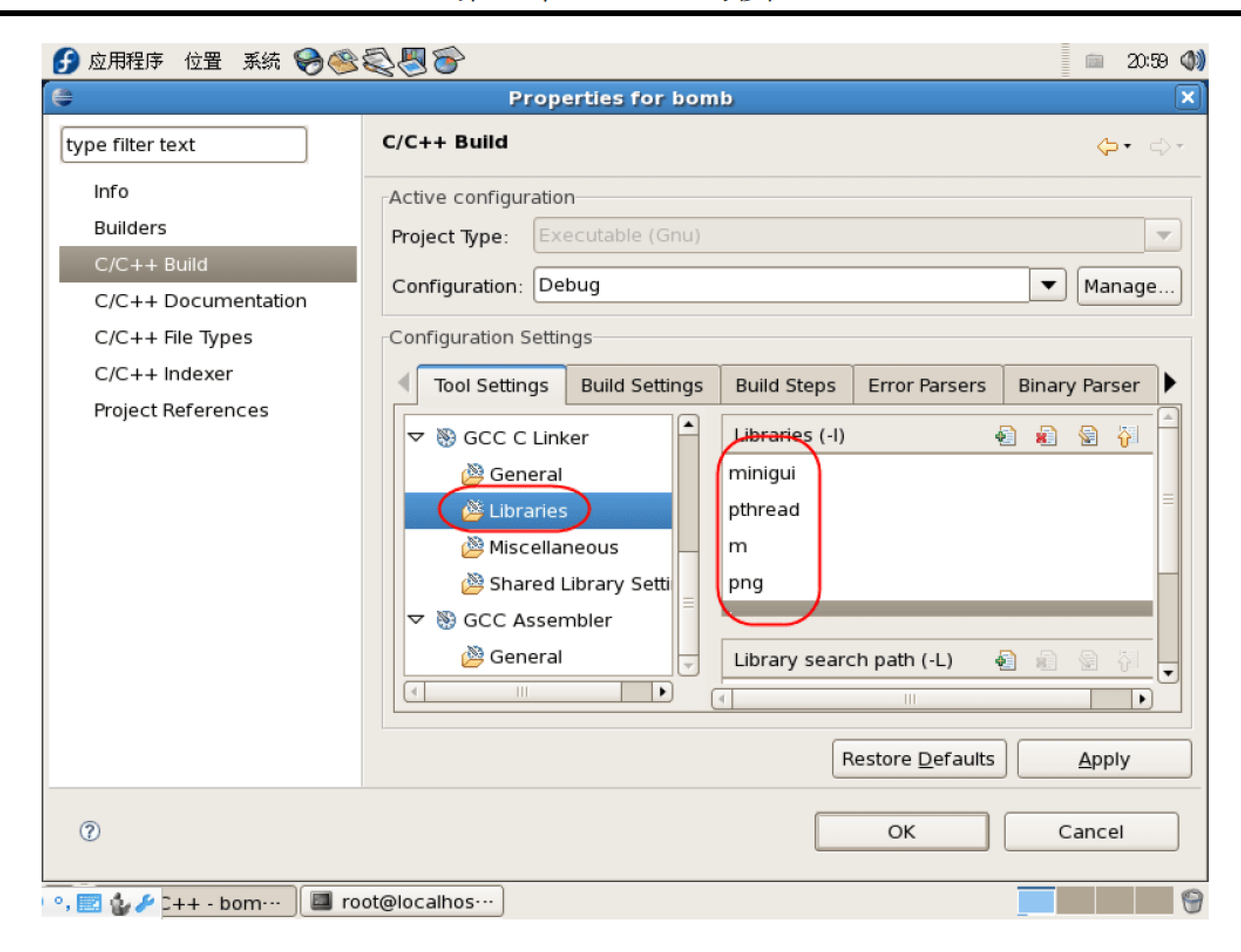



图 13.9　指定连接库文件目录

添加的库文件的路径包括：

```
/usr/local/lib
/usr/lib
/lib
```

**注意：**在配置共享目录文件 ld.so.conf 时，在末尾添加/usr/local/lib 和/usr/lib，并且使用 ldconfig 命令使其生效，否则即使配置了编译库的路径也会发生找不到共享库中的 libpng 的某个库文件，因为这些文件是通过共享库的文件进行连接过来的。例如安装 png 的库文件信息：

```
cp png.h pngconf.h /usr/local/include
chmod 644 /usr/local/include/png.h /usr/local/include/pngconf.h
cp libpng.a libpng.so.2.1.0.12 /usr/local/lib
chmod 755 /usr/local/lib/libpng.so.2.1.0.12
(cd  /usr/local/lib;  ln  -sf  libpng.so.2.1.0.12  libpng.so.2;  ln  -sf
libpng.so.2 libpng.so)
```

将 MiniGUI 的库文件、jpeg 库文件、png 库文件等目录指定连接器库包含的文件目录，如图 13.10 所示。

```
/usr/local/lib
/usr/lib
/lib
```

配置好编译器设置后，单击 Apply 和 OK 按钮后，Eclipse 会自动进行编译。如果配置正确并且没有语法错误的情况，Eclipse 会自动生成 Binary 文件。Eclipse 自动生成的 Makefile

文件在 debug 目录下与刚才编译器的设置相对应。Makefile 文件如下：

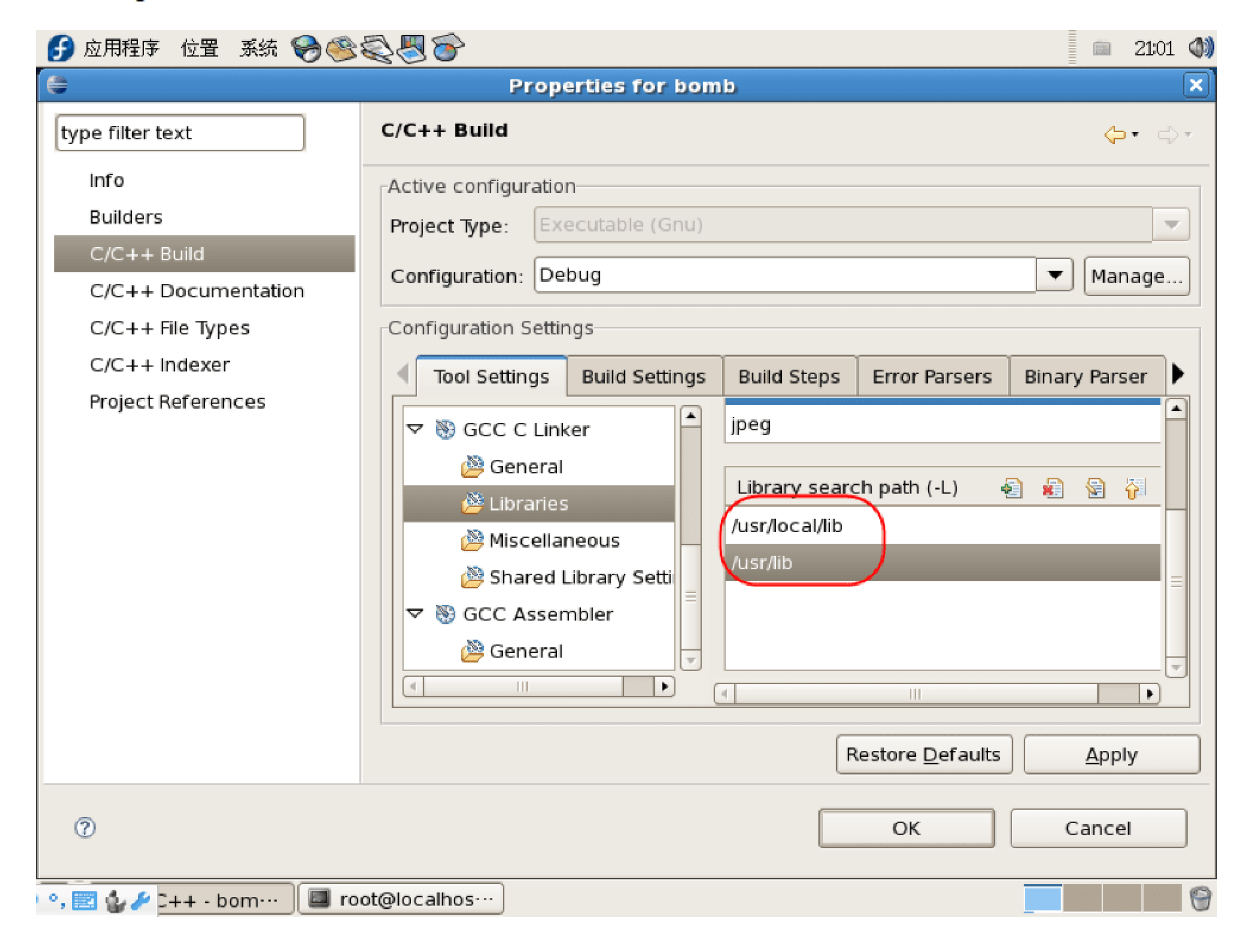



图 13.10　添加库文件路径

```
######################################################################
# Automatically-generated file. Do not edit!
######################################################################

-include ../makefile.init

RM := rm -rf             //编译之前执行清理工作

# All of the sources participating in the build are defined here
-include sources.mk      //source 指定源文件的路径
-include subdir.mk       //subdir.mk 指定编译器依赖的头文件目录
-include objects.mk
    //objects.mk 文件中指定的库文件 LIBS := -lminigui -lpthread -lm -lpng -ljpeg

ifneq ($(MAKECMDGOALS),clean)
ifneq ($(strip $(C_DEPS)),)
-include $(C_DEPS)
endif
endif

-include ../makefile.defs

# Add inputs and outputs from these tool invocations to the build variables

# All Target
all: bomb

# Tool invocations
bomb: $(OBJS) $(USER_OBJS)
```

```
    @echo 'Building target: $@'
    @echo 'Invoking: GCC C Linker'
    gcc -L/usr/local/lib -L/usr/lib -L/lib -o"bomb" $(OBJS) $(USER OBJS)
    $(LIBS)                 //指定库文件
    @echo 'Finished building target: $@'
    @echo ' '

# Other Targets
clean:                      //执行清理
    -$(RM) $(C DEPS)$(OBJS)$(EXECUTABLES) bomb
    -@echo ' '

.PHONY: all clean dependents
.SECONDARY:

-include ../makefile.targets
```

利用 Eclipse 生成的目录可以直接在命令行对工程进行编译，进入 bomb 工程所在的目录，删除 Eclipse 刚才生成的可执行文件，清理临时文件，然后执行 make，同样可以生成可执行文件。利用 Eclipse 在调试和编译程序的时候非常方便，为项目开发节省了很多时间。

```
#cd /root/workspace/bomb/Debug
#rm bomb
#make clean
#make
```

## 13.2.3　设置外部工具

在运行 MiniGUI 程序前通常要运行 qvfb，而 Eclipse 提供了这样的配置。通过选择 Run | External Tools 命令。在配置外部工具时设置工具的名字和工具的路径，如图 13.11 所示。

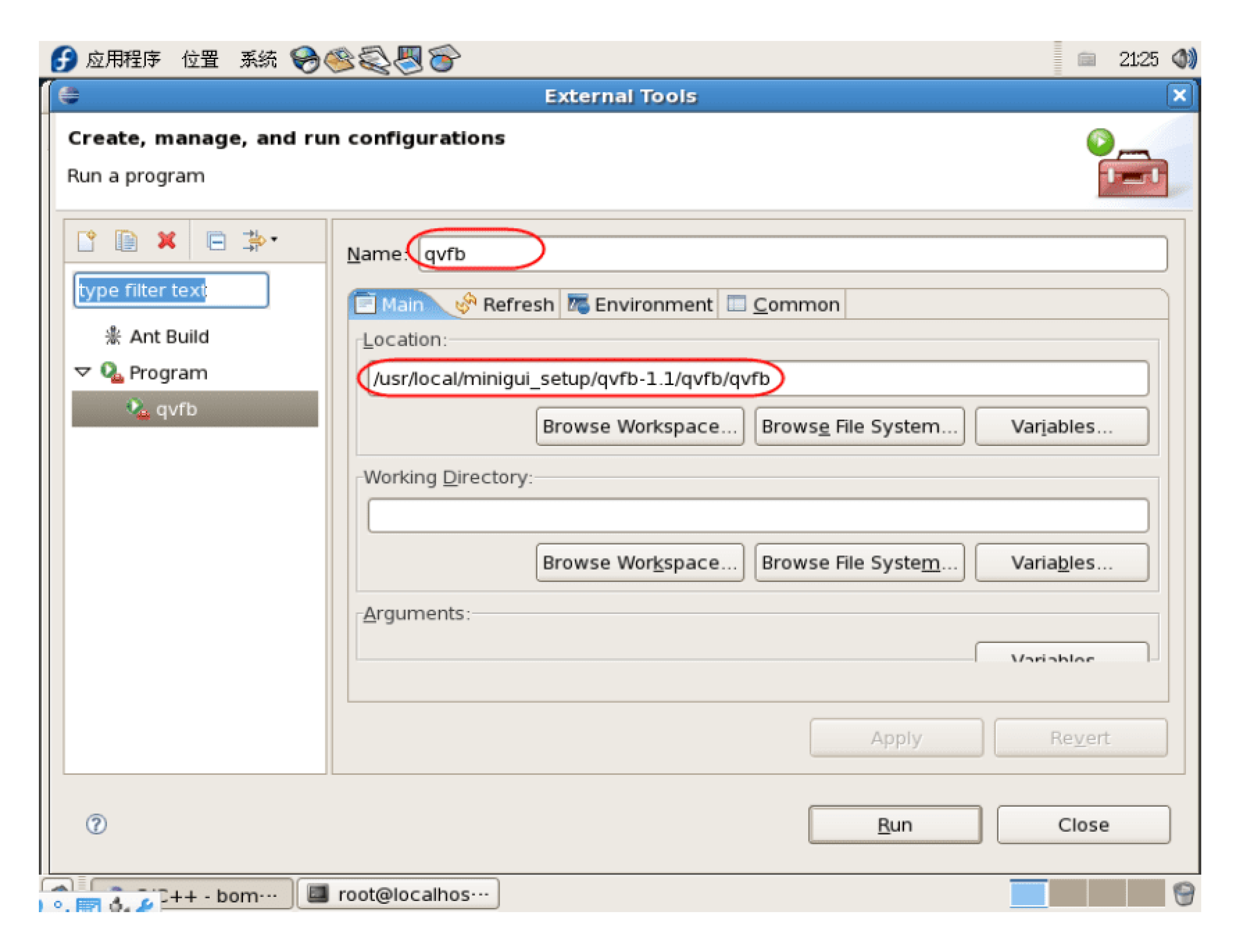



图 13.11　设置外部工具

### 13.2.4　运行程序

先运行外部工具 avfb，并设置好其显示大小 640×480 和深度为 16bit，然后右击工程，在弹出的快捷菜单中选择 Run As|Run Local C/C++ Application 命令运行程序，结果如图 13.12 所示。

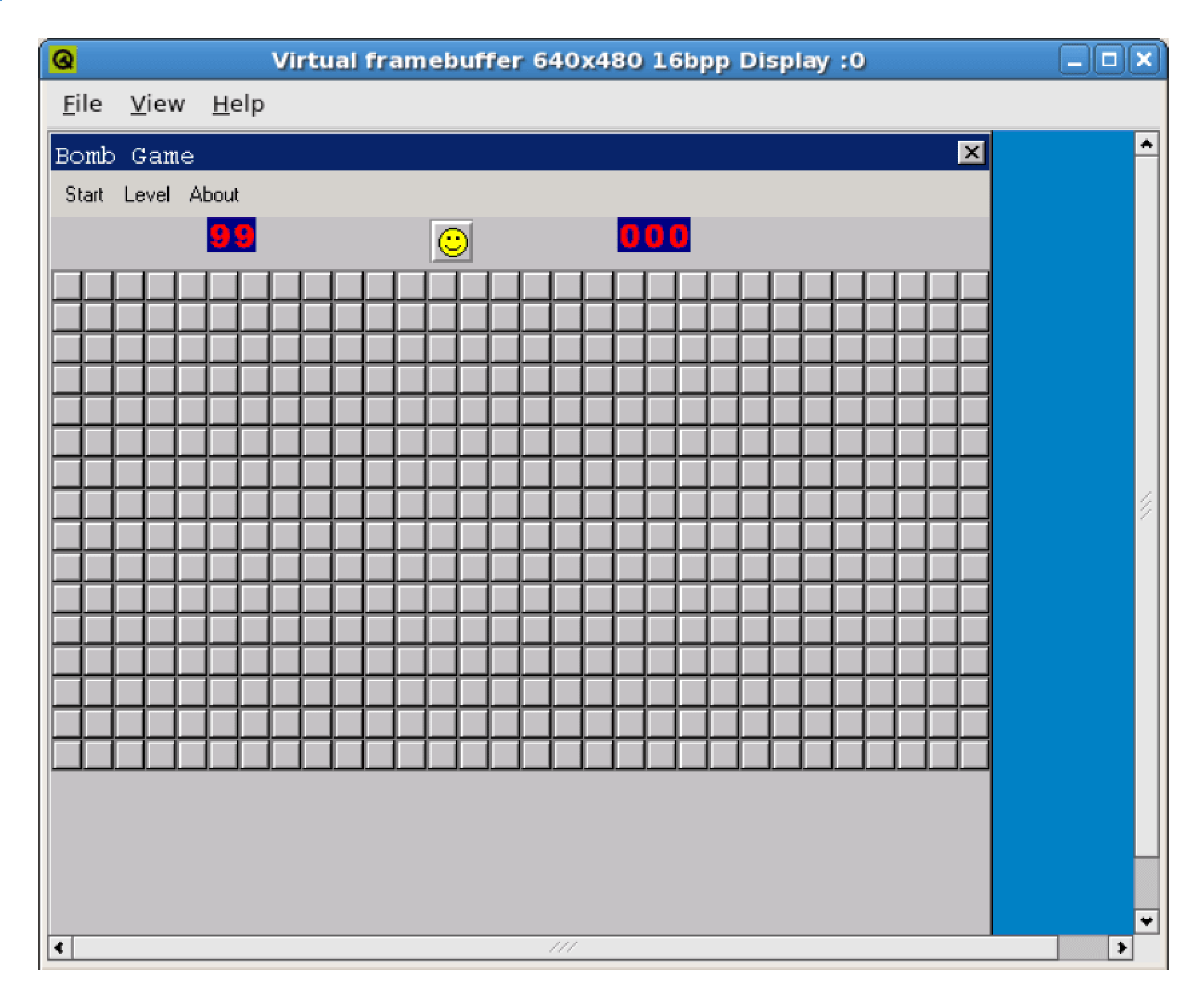



图 13.12　运行程序结果

## 13.3　VC++ 6.0 开发 MiniGUI 程序

VC++ 6.0 是很多图形界面开发者熟悉的工具，在开发图形界面和调试方面也具有很强的优势。通过 VC 6.0 调试 MiniGUI 程序给熟悉 Windows 编程读者带来了方便。

### 13.3.1　安装 Windows 开发库

可以从 MiniGUI 的官方网站获得 Windows 开发库和例子程序。本节用到的开发库为 minigui-ths-dev-2.0.4-win32.zip，例子程序为 mg-samples-2.0.4.tar.gz。

在 Windows 上直接解压两个压缩文件即可，将 mg-samples-2.0.4 文件中的.c 文件复制到解压后的 minigui-ths-dev-2.0.4-win32 目录下。使得例子程序与 dll 目录、include 目录在同一级，不这样设置也可以，在 VC 的“工程”|“设置”中包含库的路径中指向 dll 目录和 include 目录。

## 13.3.2　建立新工程

选择菜单“文件”|“新建”命令建立新工程。在工程标签页中选择工程类型为 Win32 Console Application，填写工程的名字为 spinbox，如图 13.13 所示。在建立 Win32 Console Application 第一步时选择 An empty project 单选按钮，如图 13.14 所示。单击“完成”按钮完成新工程的建立。

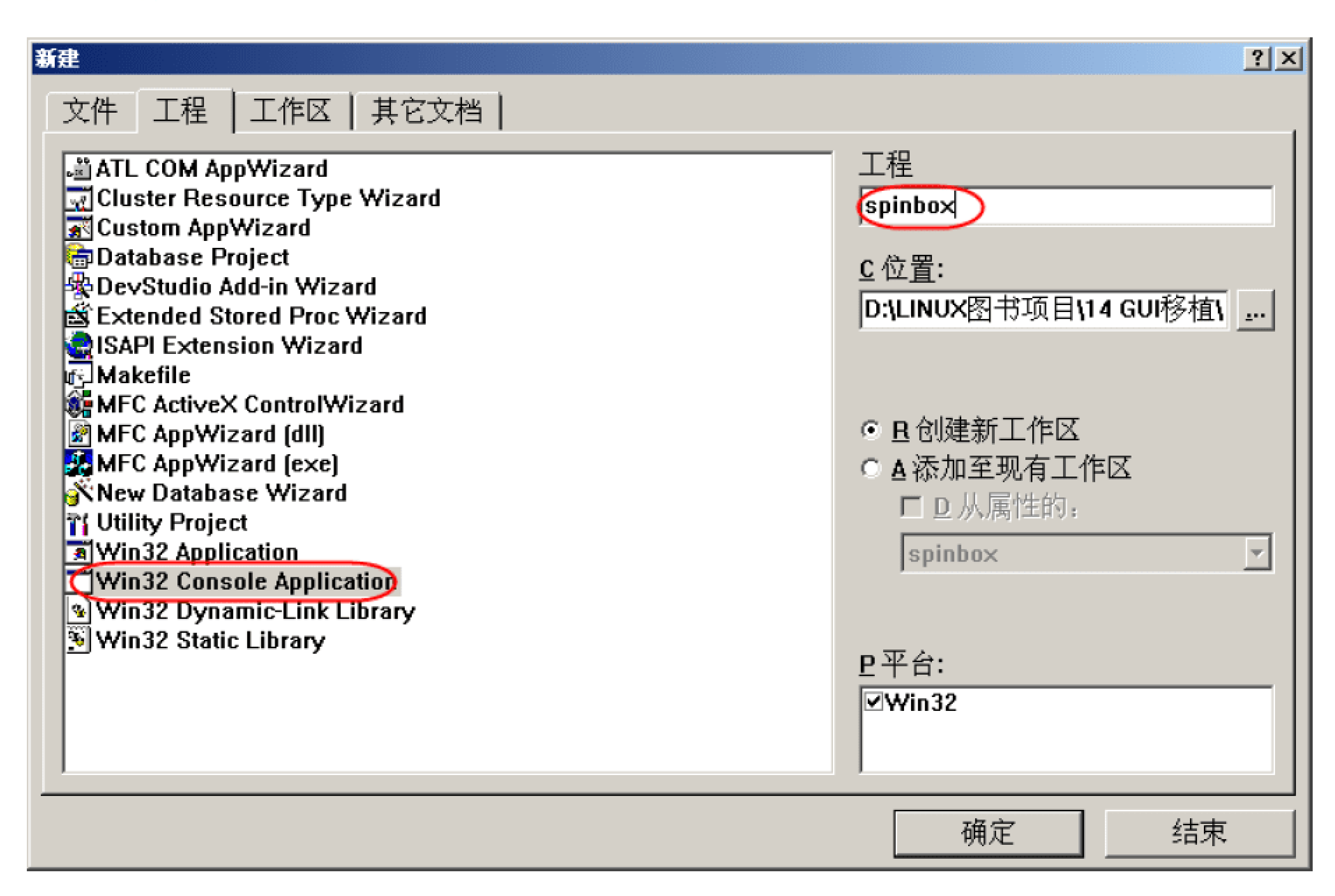



图 13.13　建立新工程

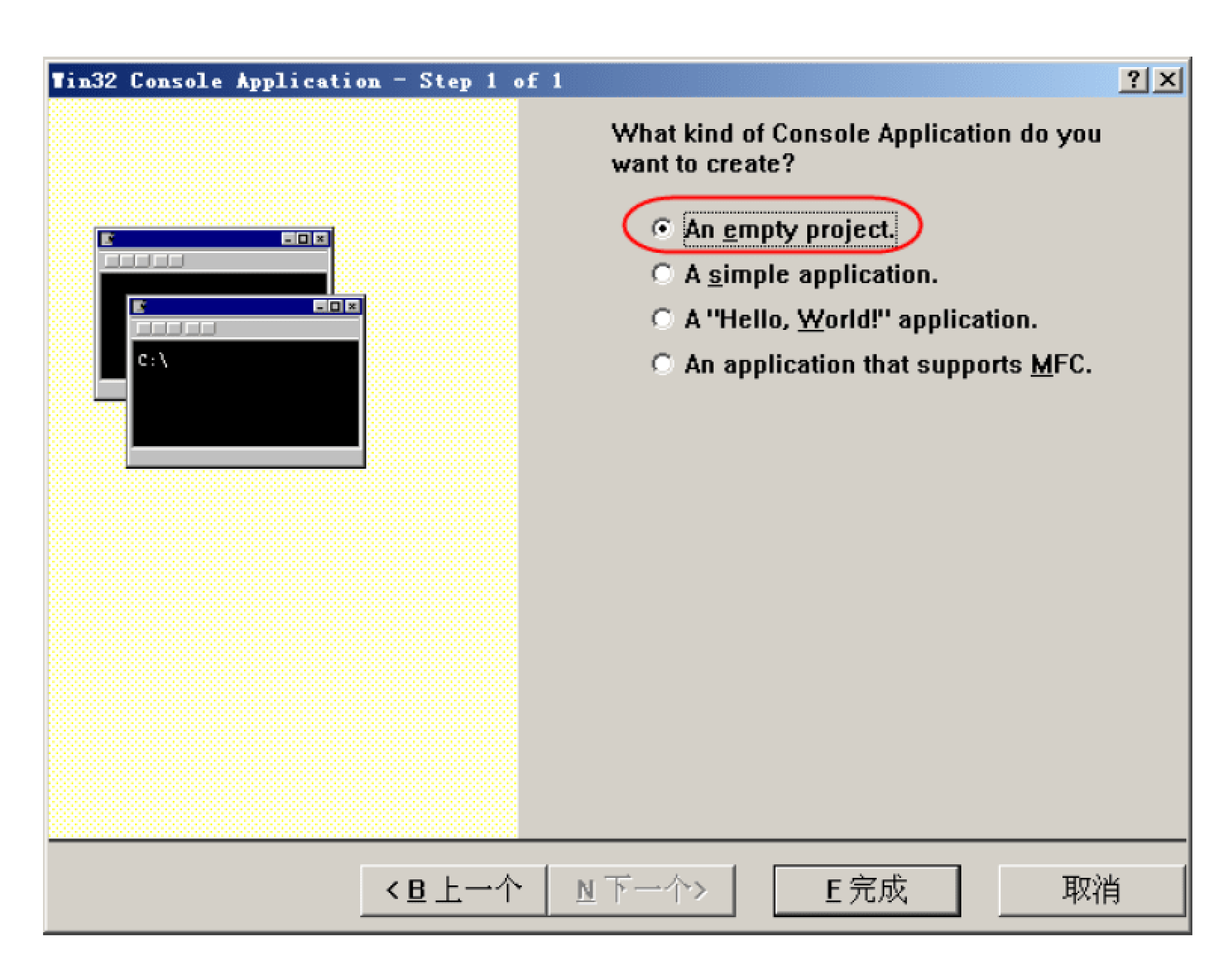



图 13.14　选择建立空工程

## 13.3.3　添加文件和设置工程

该节主要介绍 VC 的编译选项的设置，与其他编译器的设置比较类似，包括添加文件的方式、设置其输出文件路径、预处理的路径设置和编译器链接库和库的路径设置。

（1）将工作区设置为文件视图（FileView），从菜单上选择“工程”|“添加工程”|Files 命令添加源文件到工程中。

（2）选择菜单“工程”|“设置”命令，在弹出的对话框中选择 General 选项卡，设置输出文件的路径为../dll，将输出文件指定到 dll 目录下。这样在执行生成的 exe 文件时可以在执行目录下找到需要的 dll 文件，而不必将 dll 文件安装在系统目录下，或者在工程路径中指定包含 dll 的目录，如图 13.15 所示。

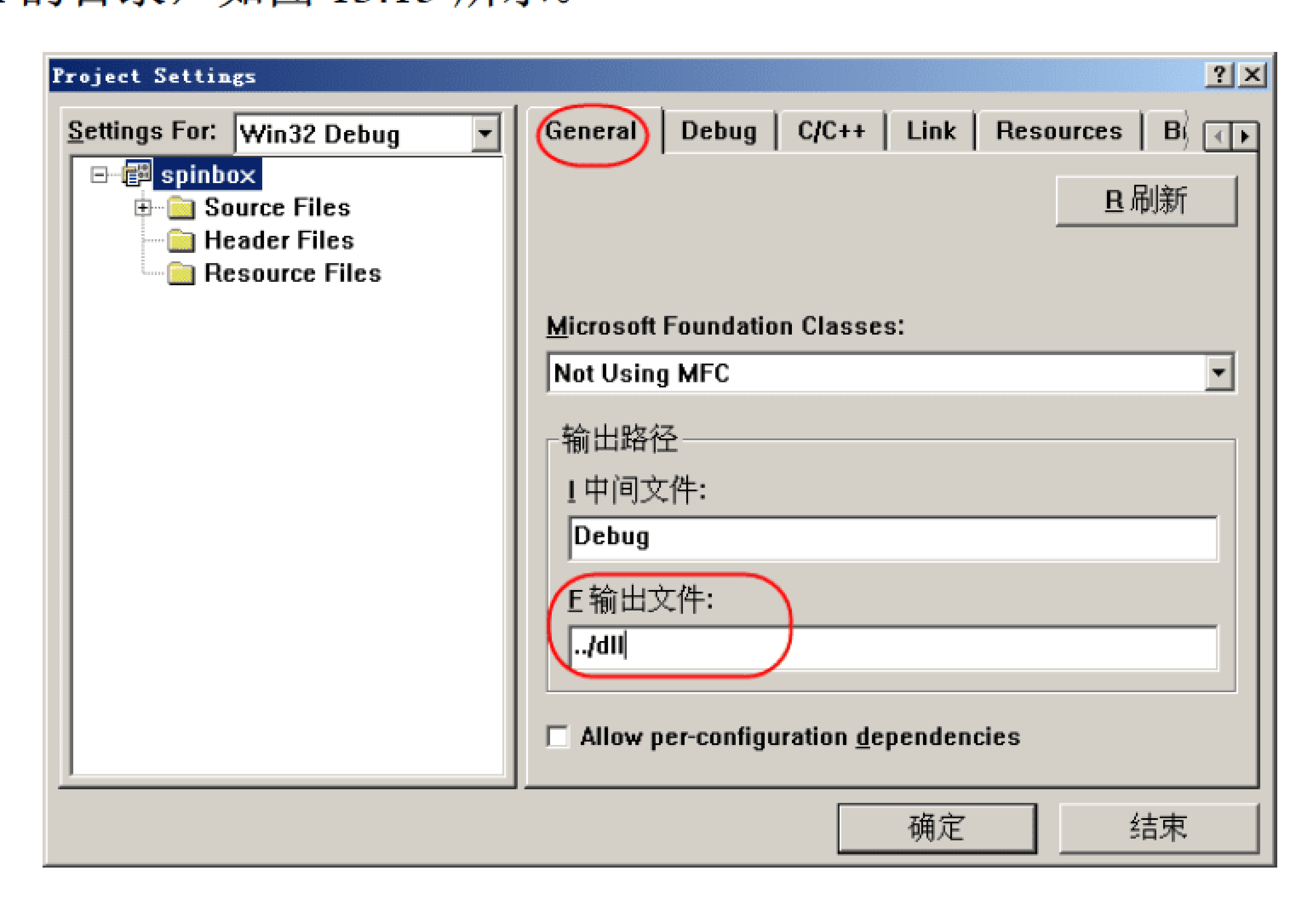



图 13.15　输出文件路径设置

（3）在 C/C++选项卡中设置分类为 Preprocessor（预处理），设置附加包含路径为../include，如图 13.16 所示。

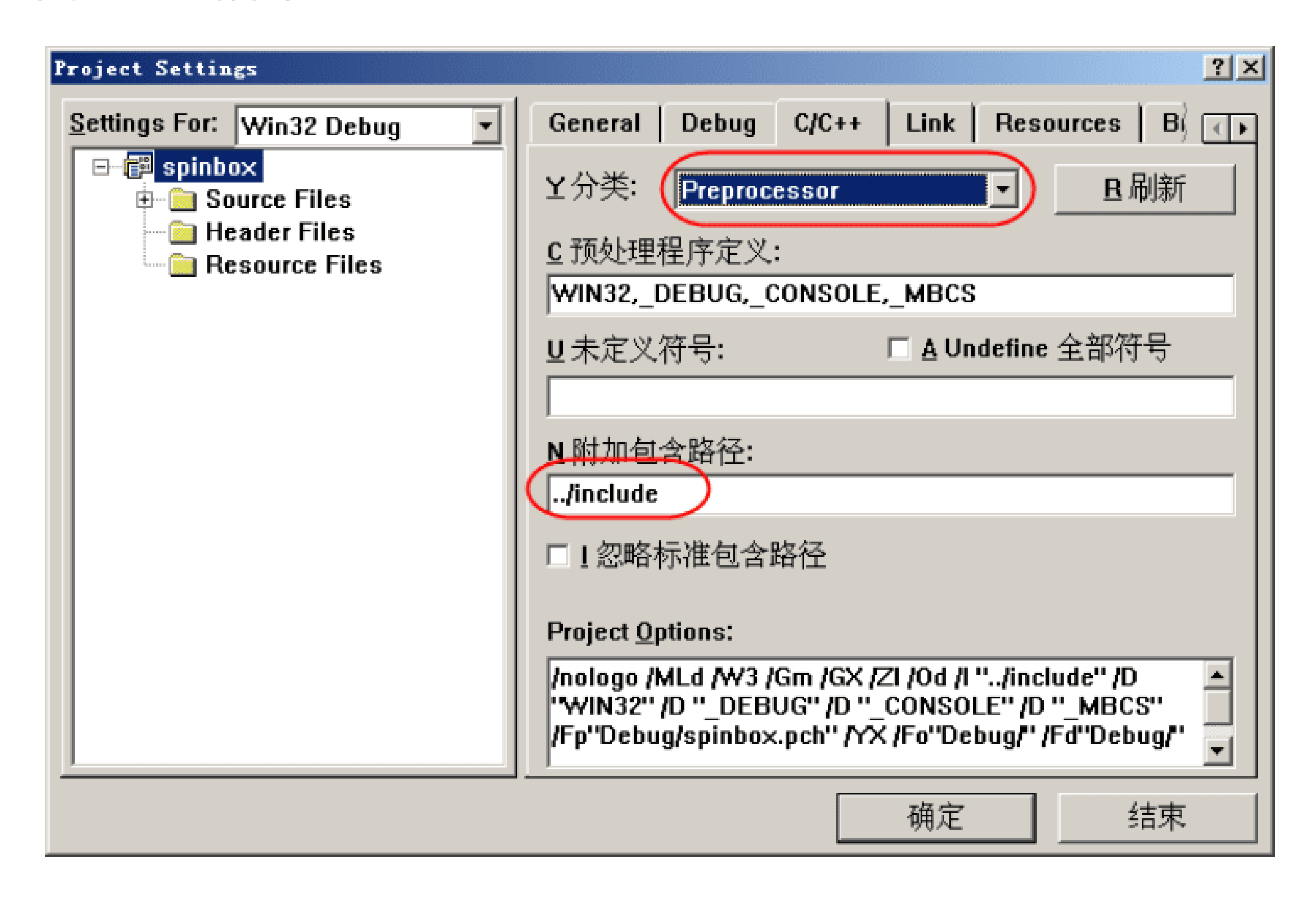



图 13.16　设置预处理的路径

（4）设置链接库和相应的路径。在 Link 选项卡中指定分类为 Input，在对象/库模块中添加所有的库（为了省去因缺少库而调试过程），将附加库路径设置为：../dll，如图 13.17 所示。添加的库文件包括 pthreadVC1.lib minigui.lib dirent.lib libjpeg.lib libpng.lib libz.lib，在 dll 目录下能找到所有对应的文件。

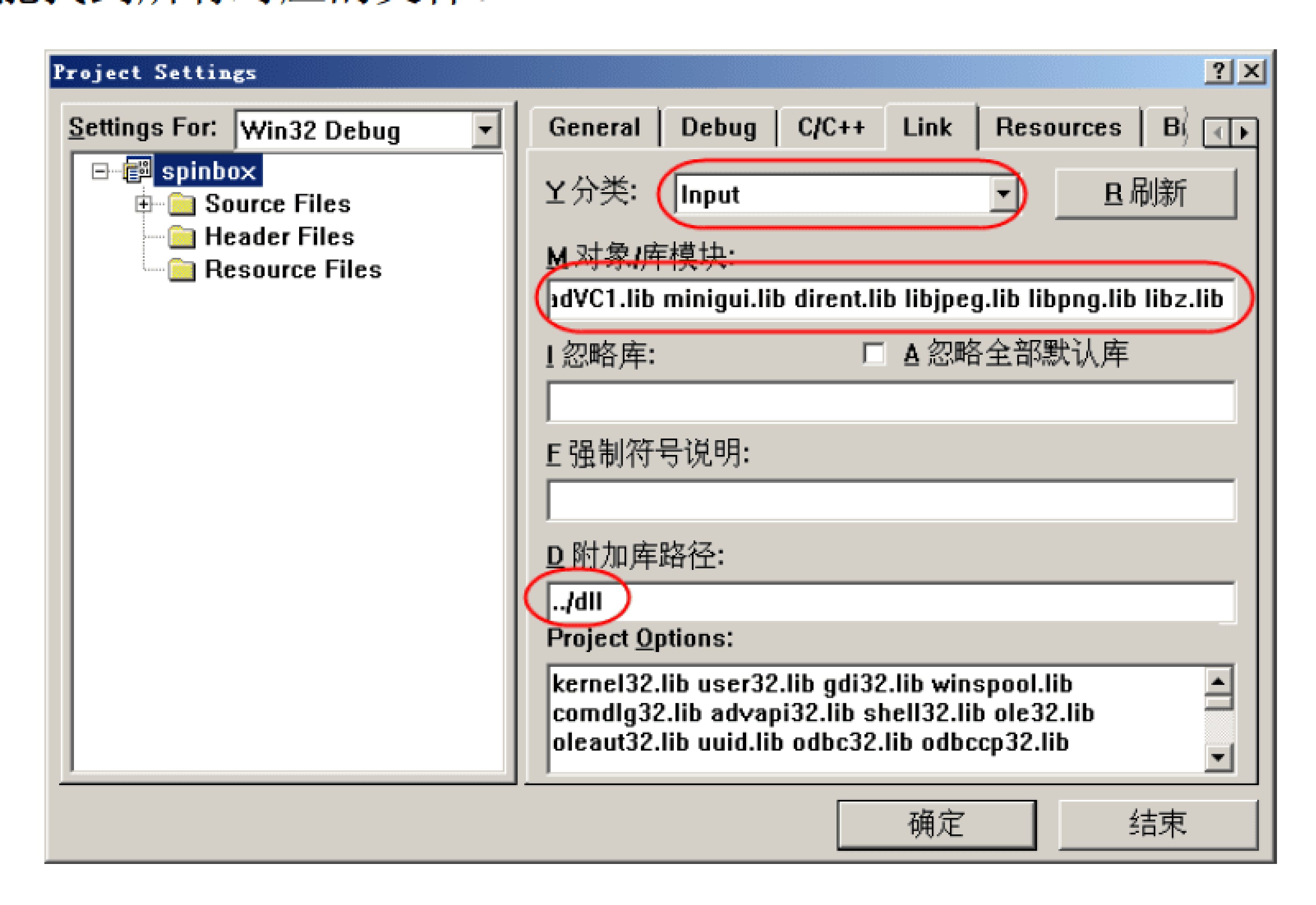



图 13.17 设置链接库和对应的路径

## 13.3.4 编译和运行程序

在 Windows 下运行的步骤类似在 Linux 中运行的步骤。选择菜单“编译”|“重建全部”命令，如果没有编译和链接错误生成可执行程序。在运行程序前先执行 wvfb 目录下的 wvfb.exe 文件，然后运行生成的可执行文件，运行的后控制台信息如图 13.18 所示，而 wvfb 则由全黑变为如图 13.19 所示。

```
"D:\linux图书项目\14 GUI移植\MiniGUI\minigui-2.0.4\minigui-ths-dev-2.0.4-win32\dl...
  5: qpf-helvetica-rrncnn-5-10-ISO8859-1,ISO8859-15,UTF-8,UTF-16LE, charsetname:
 ISO8859-1, style: 0
  6: qpf-helvetica-rrncnn-5-10-ISO8859-1,ISO8859-15,UTF-8,UTF-16LE, charsetname:
 ISO8859-15, style: 0
  7: vbf-SansSerif-rrncnn-11-13-ISO8859-1, charsetname: ISO8859-1, style: 0
  8: vbf-Courier-rrncnn-8-13-ISO8859-1, charsetname: ISO8859-1, style: 0
========== End of SBDevFonts =========

============= MBDevFonts ============
  9: rbf-fixed-rrncnn-12-12-GB2312-0, charsetname: GB2312-0, style: 0
  10: qpf-micro-rrncnn-4-4-ISO8859-1,ISO8859-15,UTF-8,UTF-16LE, charsetname: UTF
-8, style: 0
  11: qpf-micro-rrncnn-4-4-ISO8859-1,ISO8859-15,UTF-8,UTF-16LE, charsetname: UTF
-16LE, style: 0
  12: qpf-helvetica-rrncnn-5-10-ISO8859-1,ISO8859-15,UTF-8,UTF-16LE, charsetname
: UTF-8, style: 0
  13: qpf-helvetica-rrncnn-5-10-ISO8859-1,ISO8859-15,UTF-8,UTF-16LE, charsetname
: UTF-16LE, style: 0
========== End of MBDevFonts =========
system font 0: rbf-fixed-12-ISO8859-1
system font 1: *-fixed-16-GB2312
system font 2: *-Helvetica-16-GB2312
---------------------------------win_IALInit
IAL: Use wvfb engine.
```

图 13.18 控制台打印信息

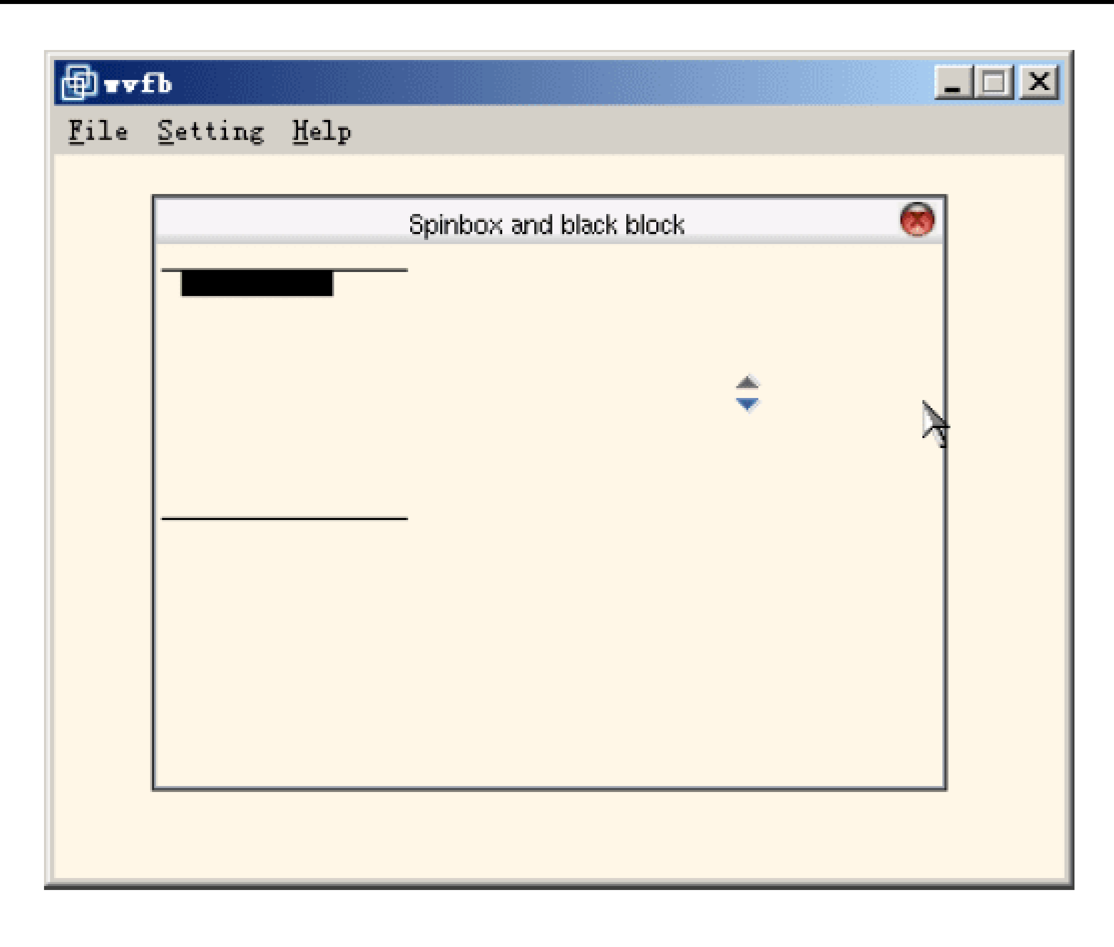



图 13.19　wvfb 窗口运行结果

## 13.3.5　MiniGUI 程序编程风格举例

MiniGUI 的编程风格和 Windows 的编程风格非常类似。包括其命名风格、编程风格等，对熟悉 Windows 编程的读者来说非常容易适应。下面为 spinbox.c 的主要代码。

```
//响应消息处理函数
static int
SpinProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    HWND hSpin;
    SPININFO spinfo;

    hSpin = GetDlgItem (hDlg, IDC_SPIN);

    switch (message)
    {

    case MSG_INITDIALOG:          //LPARAM 传递的是对话框消息
    {
        spinfo.min = 0;
        spinfo.max = 10;
        spinfo.cur = 0;
        SendMessage (hSpin, SPM_SETTARGET, 0, (LPARAM)hDlg);
        SendMessage (hSpin, SPM_SETINFO, 0, (LPARAM)&spinfo);
    }
    break;

    case MSG_KEYDOWN:             //WPARAM 传递的是鼠标按下的消息
    {
        if (wParam == SCANCODE_CURSORBLOCKUP ||
                    wParam == SCANCODE_CURSORBLOCKDOWN) {
            if (!(lParam & KS_SPINPOST)) {
                int cur;
                cur = SendMessage (hSpin, SPM_GETCUR, 0, 0);
                if (wParam == SCANCODE_CURSORBLOCKUP)
                     cur --;
                else
                    cur ++;
                SendMessage (hSpin, SPM_SETCUR, cur, 0);
```

```
            }
            InvalidateRect (hDlg, NULL, TRUE);
        }
        else if (wParam == SCANCODE_PAGEDOWN ||
                            wParam == SCANCODE_PAGEUP) {
            if (!(lParam & KS_SPINPOST)) {
                int cur;
                cur = SendMessage (hSpin, SPM_GETCUR, 0, 0);
                if (wParam == SCANCODE_PAGEUP)
                    cur -= 4;
                else
                    cur += 4;
                if (cur < 0) cur = 0;
                else if (cur > 10) cur = 10;
                SendMessage (hSpin, SPM_SETCUR, cur, 0);
            }
            InvalidateRect (hDlg, NULL, TRUE);
        }
    }
    break;

    case MSG_PAINT:             //处理绘制窗口消息
    {
        HDC hdc;
        int x, y, w, h;
        int cur;

        cur = SendMessage (hSpin, SPM_GETCUR, 0, (LPARAM)&spinfo);
        x = 10;                 //设置窗口的大小和位置信息
        y = cur*10;
        w = 60;
        h = 10;

        if (y < 10)
            y = 10;
        else if (y > 100)
            y = 100;

        hdc = BeginPaint (hDlg);
        MoveTo (hdc, 2, 10);                //绘制直线
        LineTo (hdc, 100, 10);
        Rectangle (hdc, x, y, x+w, y+h);    //绘制矩形
        SetBrushColor (hdc, PIXEL_black);   //设置画刷颜色
        FillBox (hdc, x, y, w, h);
        MoveTo (hdc, 2, 110);
        LineTo (hdc, 100, 110);
        EndPaint (hDlg, hdc);
    }
    break;

    case MSG_CLOSE:                         //关闭窗口消息
    {
        EndDialog (hDlg, 0);
    }
    break;

    }

    return DefaultDialogProc (hDlg, message, wParam, lParam);
}
```

```
static DLGTEMPLATE DlgSpin =                    //设置窗口风格
{
    WS_BORDER | WS_CAPTION,
    WS_EX_NONE,
    100, 100, 320, 240,
    "Spinbox and black block",
    0, 0,
    1, NULL,
    0
};

static CTRLDATA CtrlSpin[] =
{
    {
        CTRL_SPINBOX,
        SPS_AUTOSCROLL | WS_BORDER | WS_CHILD | WS_VISIBLE,
         230, 50, 0, 0,
        IDC_SPIN,
        "",
        0
    }
};

int MiniGUIMain (int argc, const char* argv[])    //入口函数对应 WinMain
{
/*JoinLayer 是 MiniGUI-Processes 模式的专有函数，因此包含在_MGRM_PROCESSES 的天
剑编译中。在 MiniGUI-Processes 的运行模式下，每个 MiniGUI 客户端程序在调用其他
MiniGUI 函数之前必须调用该函数将自己添加到一个层中（或创建一个新层）*/
#ifdef _MGRM_PROCESSES
    JoinLayer(NAME_DEF_LAYER , "spinbox" , 0 , 0);
#endif

    if (!InitMiniGUIExt()) {
        return 2;
    }

    DlgSpin.controls = CtrlSpin;

    DialogBoxIndirectParam (&DlgSpin, HWND_DESKTOP, SpinProc, 0L);

    MiniGUIExtCleanUp ();

    return 0;
}
```

## 13.4　MiniGUI 的交叉编译和移植

MiniGUI 作为遵循 GPL 条款发布的自由软件，其目标是为基于 Linux 的实时嵌入式系统提供一个轻量级的图形用户界面支持系统。与 QT/Embedded、MicoroWindows 等其他 GUI 相比，MiniGUI 的最显著特点就是轻型、占用资源少。本节将介绍 MiniGUI 移植到 mini2440 上的过程。

### 13.4.1　交叉编译 MiniGUI

在 13.1 节中已经介绍了 MiniGUI 在 PC 上安装的过程，下载的软件包和相关的地址请参考 13.1 节。交叉编译 MiniGUI 主要包括交叉编译 libminigui、安装 minigui-res 和交叉编译演示程序 mg-samples。

（1）进行 MiniGUI 函数库的编译和安装。解压 libminigui-1.6.10.tar.gz 软件包，进入该目录，运行./configure 脚本。

```
#CC=arm-linux-gcc \
./configure --prefix=/usr/local/arm/4.3.2/arm-none-linux-gnueabi/ \
                           //指定编译生成的库存放的路径
--build=i386-linux \
--host=arm-linux \
--target=arm-linux
```

生成定制的 Makefile 文件，然后可以继续执行 make 和 make install 命令编译并安装 libminigui，安装成功后，MiniGUI 的函数库和头文件及配置文件等资源将被安装到/usr/local/arm/4.3.2/arm-none-linux-gnueabi 目录中。函数库安装在 lib/子目录中；头文件安装在 include/子目录中；手册被装在 man/子目录中；配置文件被装在 etc/子目录中。

```
#make
#make install
```

（2）MiniGUI 资源的编译安装。解压 minigui-res-1.6.10.tar.gz，进入 minigui-res-1.6.10 目录。值得注意的是，在执行 make install 操作之前，修改目录中的 configure.linux 文件。打开 configure.linux 文件，prefix 选项部分的默认值为$(TOPDIR)/usr/local，需要将这里修改为 prefix=$(TOPDIR)/usr/local/arm/4.3.2/arm-none-linux-gnueabi/，读者在这里修改为本机的交叉编译器路径，这样执行 make install 操作之后，安装脚本会自动把 MiniGUI 资源文件安装到/usr/local/arm/4.3.2/arm-none-linux-gnueabi/lib/minigui/res/目录下。

（3）交叉编译 MiniGUI 的演示程序。交叉编译 MiniGUI 的演示程序。解压 mg-samples-str-1.6.10.tar.gz，进入 mg-samples-str-1.6.10 目录执行脚本：

```
#CC=arm-linux-gcc           //指定编译器为交叉编译器 arm-linux-gcc
CFLAGS=-I/usr/local/arm/4.3.2/arm-none-linux-gnueabi/include
                           //指定依赖的头文件路径
LDFLAGS=-L/usr/local/arm/4.3.2/arm-none-linux-gnueabi/lib
                           //指定依赖的库文件路径
./configure
--build=i386-linux
--host=arm-linux
--target=arm-linux
```

执行上述命令生成 Makefile 文件，继续执行 make 操作，完成演示程序的编译。

```
#make
```

编译完成后，可以使用 readelf 查看生成的演示程序。进入 src 子目录，查看文件头信息。

```
 #readelf -h combobox
```

正确编译后，Machine 字段为 ARM，其文件头信息如下：

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x87dc
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5672 (bytes into file)
  Flags:                     0x5000002, has entry point, Version5 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

## 13.4.2　移植 MiniGUI 程序

移植 MiniGUI 程序前，文件系统加入 LCD 驱动，才能在 LCD 上查看 MiniGUI 运行结果。另外，内核也需要添加对图形引擎的支持。

（1）对内核的配置。选择 Device Drivers ---> Graphics support ---> Support for frame buffer devices --->配置对图形引擎的支持，如图 13.20 所示。选择 Device Drivers ---> Graphics support ---> Console display driver support ---> Framebuffer Console support 支持控制台帧缓存，如图 13.21 所示。

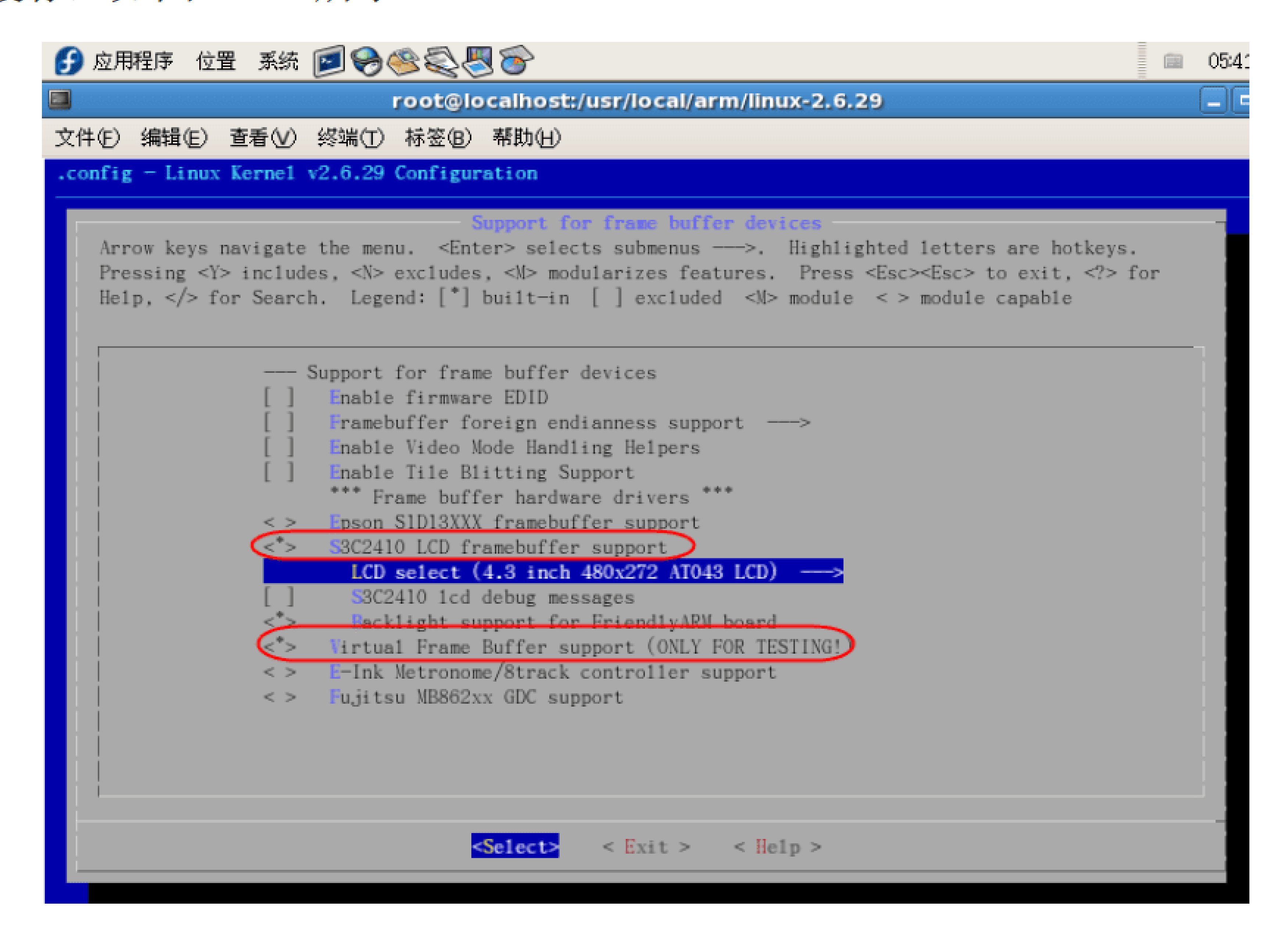



图 13.20　配置内核对图形引擎的支持

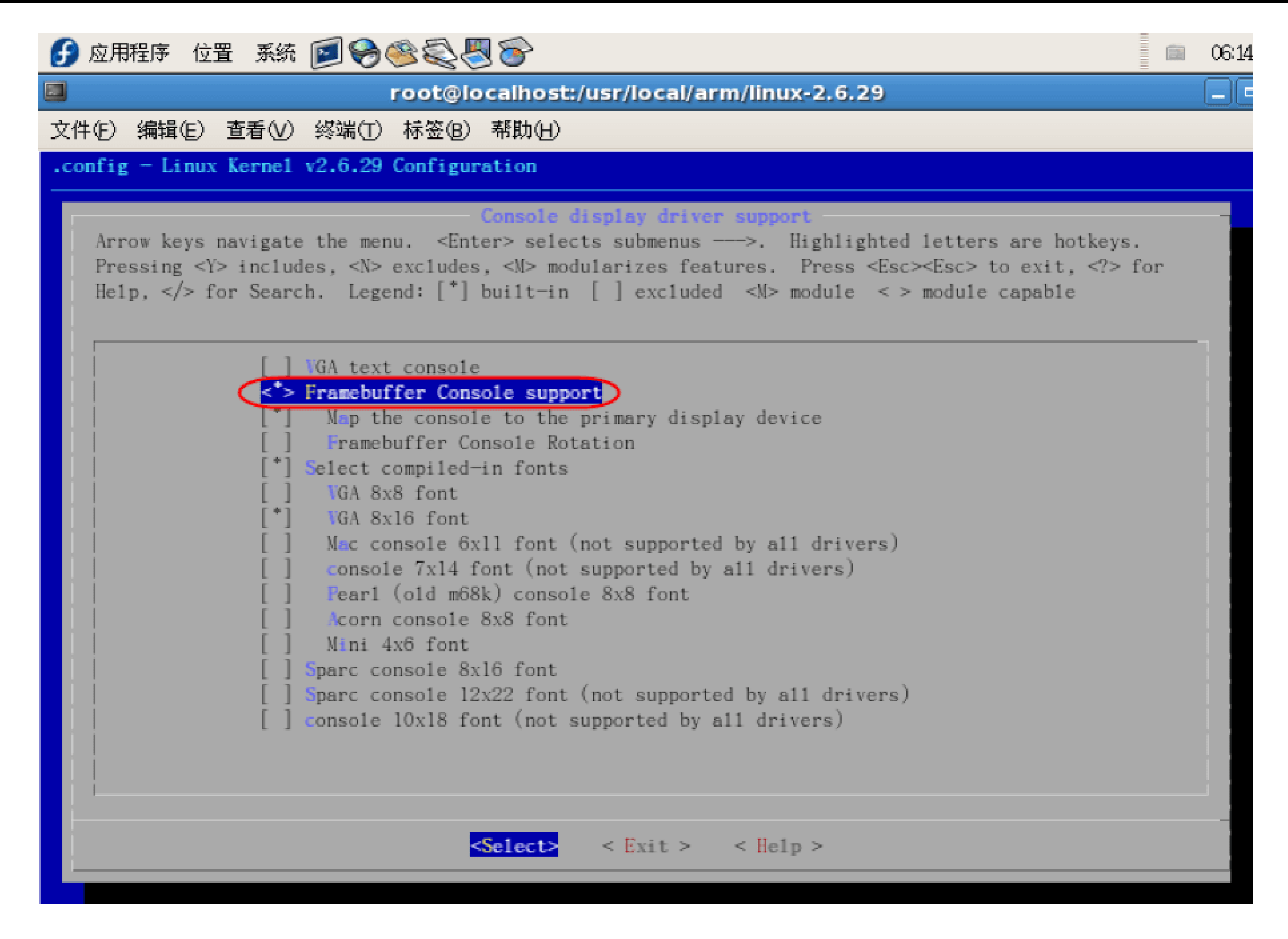



图 13.21　配置支持控制台帧缓存

（2）加入对 LCD 的驱动，这部分内容在后面会详细介绍移植过程。这里使用开发板供应商提供的文件系统支持 LCD 驱动。

（3）下载 MiniGUI 配置文件到开发板。将配置文件 MiniGUI.cfg 复制到 mini2440 开发板文件系统的/usr/local/etc/目录下。对配置文件进行如下修改：

```
vi /usr/local/arm/4.3.2/arm-none-linux-gnueabi/etc/MiniGUI.cfg
[system]
# GAL engine and default options
gal engine=fbcon
defaultmode=480x272-16bpp

# IAL engine
ial engine=console
mdev=/dev/input/mice
mtype=IMPS2

[fbcon]
defaultmode=480x272-16bpp

[qvfb]
defaultmode=480x272-16bpp
display=0
```

（4）将/usr/local/arm/4.3.2/arm-none-linux-gnueabi/lib 目录下的下列库文件下载到/usr/local/lib 目录下。/usr/local/lib 目录下的库文件包括：

```
libmgext-1.6.so.10  libmgext.so  libminigui.la  libvcongui-1.6.so.10.0.0
libmgext-1.6.so.10.0.0 libminigui-1.6.so.10  libminigui.so  libvcongui.a
libmgext.a  libminigui-1.6.so.10.0.0    libsupc++.a libvcongui.la
libmgext.la     libminigui.a    libvcongui-1.6.so.10    libvcongui.so
```

（5）将/usr/local/arm/4.3.2/arm-none-linux-gnueabi/lib/minigui 目录下的整个 res 文件夹

下载到/usr/local/lib/minigui 目录下，开发板上的这个目录是参考 MiniGUI.cfg 配置文件的目录。

（6）在开发板上运行 MiniGUI。将 mg-samples-1.6.10 目录中的 src 下生成可执行文件，下载到开发板/usr/local/bin/minigui 目录下。

## 13.5　小　　结

移植 MiniGUI 的过程，有类似于其他模块的移植部分，如交叉编译；也有不同于其他模块的部分，如需要增加对图形引擎的支持。在编译安装 MiniGUI 的过程中注意对库文件目录的安装。移植的时候注意库文件和资源文件在开发板上的存放位置。

# 第 14 章　Qt 开发与 Qtopia 移植

Qt 是一个用 C++编写的成熟的跨平台 GUI 工具包。Qt 提供给应用程序开发者大部分的功能，来完成建立合适、高效的图形界面程序与后台执行的应用程序，它提供的是一种面向对象可扩展的和基于组件的编程模式。本章主要介绍 Qt 在 PC 上的安装、编程及 Qtopia 在上位机上的安装、开发和移植到 ARM 板。本章依然遵循安装、编译、调试、交叉编译和移植，这一逐步深入的过程。

## 14.1　Qt 安装与编程

安装软件的方法一般是挂载系统安装盘，通过安装和卸载软件进行安装。如果没有系统安装文件的情况下，可以到网上下载 rpm 包进行安装。Qt 也分为免费版和商业版，免费版缺少支持且不能把 Qt 软件用于商业开发。

### 14.1.1　下载安装 Qt

Fedora 6 自带的 Qt 安装包是 qt-devel-3.3.6-13.i386.rpm。在从 XWindow 模式下直接进入 RPM 文件夹下双击安装，或者复制到/usr/local 目录下，采用下面的命令进行安装。

```
#rpm -Uvh qt-devel-3.3.6-13.i386.rpm              //安装 rpm 包
```

用户也可以从 Trolltech 公司的网站上获得 Qt 的源码包 qt-x11-free-3.3.8b.tar.gz 自己创建 Qt。编译和安装源码的方法如下：

```
#tar zxvf qt-x11-free-3.3.8b.tar.gz    //解压源码包
#cd qt-x11-free-3.3.8b
#./configure                           //执行 configure 后输入 yes 表示接受 GPL
Type 'Q' to view the Q Public License.
Type '2' to view the GNU General Public License version 2.
Type '3' to view the GNU General Public License version 3.
Type 'yes' to accept this license offer.
Type 'no' to decline this license offer.
Do you accept the terms of either license? Yes     //输入 yes 回车后开始生成
                                                     Makefile
#make                                              //编译
```

编译安装完成后，将 Qt 库路径添加到文件/etc/ld.so.conf 中：

```
/usr/lib/qt-3.3/lib                    //读者的 Qt 或许是/usr/lib/qt-3/lib，可以
                                         通过下面的命令进行检验
```

安装好后，可以使用 echo 检验环境变量 QTDIR。

```
#echo $QTDIR
/usr/lib/qt-3.3
```

通过 ldconfig 命令使之在文件/etc/ld.so.conf 中生效。

```
#ldconfig
```

如果上面的过程安装完成，14.1.2 节将带读者进入 Qt 编程，同时检验本节是否安装正确。查看安装的库的名字。

```
#ls /usr/lib/qt-3.3/lib/lib
libdesignercore.a     libqassistantclient.prl    libqt-mt.so libqui.prl
libdesignercore.prl   libqsa.so.1    libqt-mt.so.3 libqui.so
libeditor.a          libqsa.so.1.1    libqt-mt.so.3.3 libqui.so.1
libeditor.prl         libqsa.so.1.1.4    libqt-mt.so.3.3.6 libqui.so.1.0
libqassistantclient.a  libqt-mt.prl    libqt-mt.so.3.3.8 libqui.so.1.0.0
```

**注意：** 通过查看库可以知道这里安装的 Qt 库的名字为 gt-mt，读者的 Qt 的库也许叫 qt。lib 代表库，so 代表共享库，后面数字代表版本。

## 14.1.2　Qt 编程

Qt 是采用 C++编写的，其编程的单元也是类。对于熟悉 C++面向对象编程的读者来说，学习 Qt 编程将会非常容易。如果读者有 MFC 或者其他 GUI 开发经验，学习 Qt 编程会非常容易。即使读者没有学过 C++，只要读者有面向对象的基础即可。下面将通过一个简单的注册对话框介绍 Qt 编程。如果读者没有学过任何一门面向对象的开发语言，建议读者在开发 Qt 或者进行嵌入式 Qt 开发之前先学习面向对象的编程思想。

本例实现一个简单的登录对话框界面。程序主要有类的定义和类的实现两部分。Login.h 头文件定义了类的声明，声明了类的属性和操作。Login.cpp 文件完成方法的实现。下面依次介绍各个部分。

### 1．类的定义部分

类的定义部分，指定了所继承的父类，定义了构造函数，声明了 3 个 QLineEdit 类型属性和 1 个方法 Clicked()。其代码如下：

```
#include <qmainwindow.h>
#include <qlineedit.h>
#include <qstring.h>
class Login : public QmainWindow
                        //继承 QMainWindow，这样可以继承 QMainWindow 的方法
{
Q_OBJECT                //使用信号与槽机制时，类的声明中必须加上 Q_OBJECT 语句
public:
    Login(QWidget *parent = 0, const char *name = 0);
    QLineEdit *username_entry;
    QLineEdit *password_entry;
    QLineEdit *pwcheck_entry;
private slots:
    void Clicked();
};
```

### 2．类的实现部分

构造函数设置了对话框中控件的显示风格，指定了窗口的大小，定义了信号与槽的连接关系。其代码如下：

```
#include "Login.moc"                      //运行预处理头文件得到的文件
#include <qpushbutton.h>
#include <qapplication.h>
#include <qlabel.h>
#include <qlayout.h>
#include <iostream>
Login::Login(QWidget *parent, const char *name) :QMainWindow(parent, name)
{
    QWidget *widget = new QWidget(this);
    setCentralWidget(widget);
    QGridLayout *grid = new QGridLayout(widget,4,2,10, 10,"grid");
                                          //将对话框分为 4 行 2 列
    username_entry = new QLineEdit( widget, "username_entry");
    password_entry = new QLineEdit( widget, "password_entry");
    pwcheck_entry = new Login( widget, "pwcheck_entry");
    password_entry->setEchoMode(QLineEdit::Password);
                                          //回显时为星号
    pwcheck_entry->setEchoMode(QLineEdit::Password);

    grid->addWidget(new QLabel("Username", widget, "userlabel"),
                0, 0, 0);                 // Username 为第 1 行第 1 列(0,0)
    grid->addWidget(new QLabel("Password", widget, "passwordlabel"),
                1, 0, 0);                 // Password 为第 2 行第 1 列(1,0)
    grid->addWidget(new QLabel("PasswordCheck", widget, "pwcheck_entry"),
                2, 0, 0);                 // Password 为第 3 行第 1 列(2,0)

    grid->addWidget(username_entry, 0,1, 0);
    grid->addWidget(password_entry, 1,1, 0);
    grid->addWidget(pwcheck_entry, 2,1, 0);

    QPushButton *button = new QPushButton ("Ok", widget, "button");
                                          //定义按钮
    grid->addWidget(button, 3,1,Qt::AlignRight);
    resize( 350, 200 );
    connect (button, SIGNAL(clicked()), this, SLOT(Clicked()));
                                          //连接信号与槽，响应函数为 Clicked()
}

void Login::Clicked(void)                 //单击 Ok 按钮时，Clicked 响应
{
    std::cout <<"password:"<< password_entry->text() << "\n";
    std::cout <<"pwcheck:"<< pwcheck_entry->text() << "\n";
}
```

### 3．主函数部分

主函数是程序的入口点，创建一个 Login 类实例，显示窗口并启动事件循环机制，当有事件（Clicked()）发生时响应事件机制（将 password 和 pwcheck 输出到控制台）。其代码如下：

```
int main(int argc, char **argv)
{
    QApplication app(argc,argv);
    Login *window = new Login();
    app.setMainWidget(window);          //设置应用程序的主窗口部件
    window->show();
    return app.exec();                  //启动事件循环
}
```

执行下面命令进行编译和运行，运行结果如图 14.1 所示。

```
# moc Login.h -o Login.moc
# g++ -o login Login.cpp -I$QTDIR/include -L$QTDIR/lib -lqt-mt
#./login
```

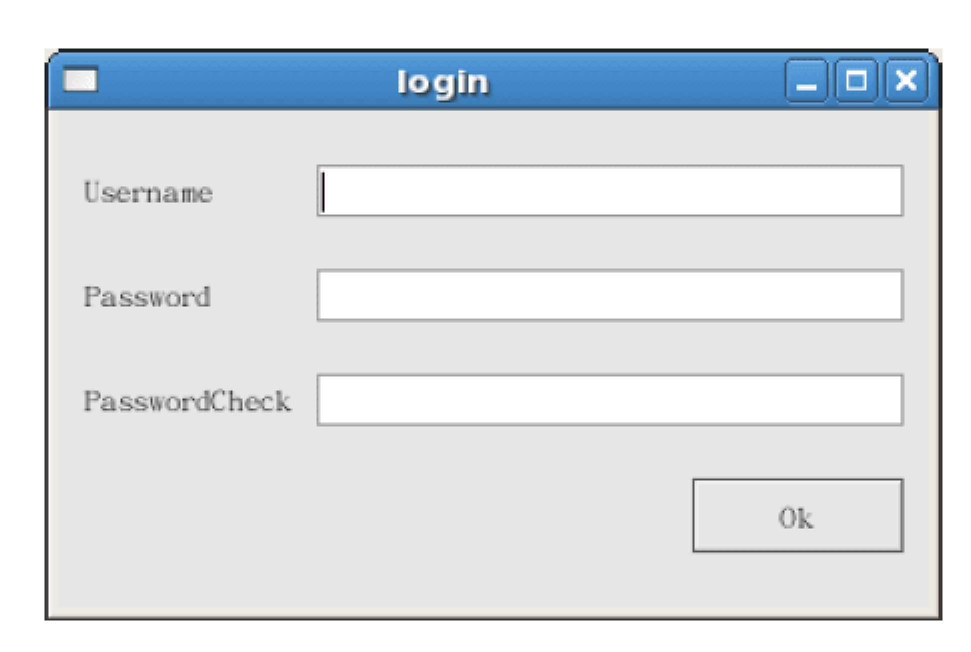



图 14.1　login 运行结果

## 14.1.3　使用 qmake 生成 Makefile

使用 qmake 生成 Makefile 也是 Qt 编程的基础。在 Qt 集成开发工具中将 qmake 嵌入到开发工具中辅助开发人员生成 Makefile 文件。下面通过一个例子程序说明如何使用 qmake 生成 Makefile。将 qmake 所在的路径加入到环境变量 PATH 中，使用下面命令设置 PATH。

```
#export PATH=/usr/lib/qt-3.3/bin:$PATH
```

通过一个简单的例子说明 qmake 如何生成 Makefile 文件，新建 hello.c 文件。

```
#vi hello.c
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

编写项目文件 hell0.pro，该文件用于 qmake 生成 Makefile 文件。

```
#vi hello.pro
SOURCES = hello.c
CONFIG += qt warn_on release
```

qmake 可以通过下面的命令生成 Makefile，然后直接 make 可以生成可执行程序。

```
#qmake -o Makefile hello.pro
#make
#./hello
```

## 14.2　Qtopia Core 在 X86 平台上的安装和应用

Qtopia Core 是 QT 的嵌入式版本，是 Trolltech 公司从版本 4.1 开始将 Qt/E 并入 Qtopia 产品线产生的结果。Qtopia Core 与 Qt/X11 的最大区别就是不依赖 X Server 或 Xlib，而是直接访问帧缓冲设备（Frambuffer），这样做最显著的特点就是减少了内存的消耗。安装在 X86 平台方便开发和调试，调试验证后移植嵌入式环境。

### 14.2.1　Qtopia Core 安装准备

在 X86 平台上编译 Qtopia Core 时需要准备两个程序：

- qt-x11-opensource-src-4.2.2.tar.gz，编译其目的主要为了获得帧缓冲（QVFB）。
- qtopia-core-opensource-src-4.3.5.tar.gz，编译、安装获得环境 QtopiaCore-4.3.5-arm。

将 qt-x11-opensource-src-4.2.2.tar.gz 和 qtopia-core-opensource-src-4.3.5.tar.gz 赋值到/usr/local/arm/mini2440 目录下。

```
#tar zxvf qt-x11-opensource-src-4.2.2.tar.gz
#cd qt-x11-opensource-src-4.2.2
```

如果读者不是第一次安装 Qt，在编译之前可以检查本机是否有旧版本，将 QTDIR 指向新安装的路径下。设置 QTDIR、PATH 和 LD_LIBRARY_PATH 这 3 个环境变量。

```
# export QTDIR=$PWD                //将 QTDIR 设置为当前目录
# export PATH=$QTDIR/bin:$PATH
# export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

完成环境变量设置后，使用 echo 命令检验环境变量是否设置成功。接下来按照下面的命令依次配置、编译、安装 qt-x11-opensource-src-4.2.2 和安装 qvfb。

```
#./configure
#yes
#gmake
#gmake install
#cd tools/qvfb
#make                              //进入 qvfb 目录下，编译 qvfb 工具
```

qvfb 安装成功后，在目录/usr/local/arm/mini2440/qt-x11-opensource-src-4.2.2/bin 下会生成 qvfb、uic、moc、designer 等工具。在该目录下测试 qvfb 如图 14.2 所示，测试 designer 如图 14.3 所示。

```
#cd /usr/local/arm/mini2440/qt-x11-opensource-src-4.2.2/bin
#./qvfb
#./designer                        //启动 Qt designer 工具
```

到目前为止已经可以进行界面设计，对可执行程序可以在 X86 平台上进行虚拟仿真了。通过 Qt designer 工具进行界面设计保存为.ui 文件，通过 uic 工具将其转为.h 文件，在使用该文件的工程中就可以将该.h 文件包含进来。

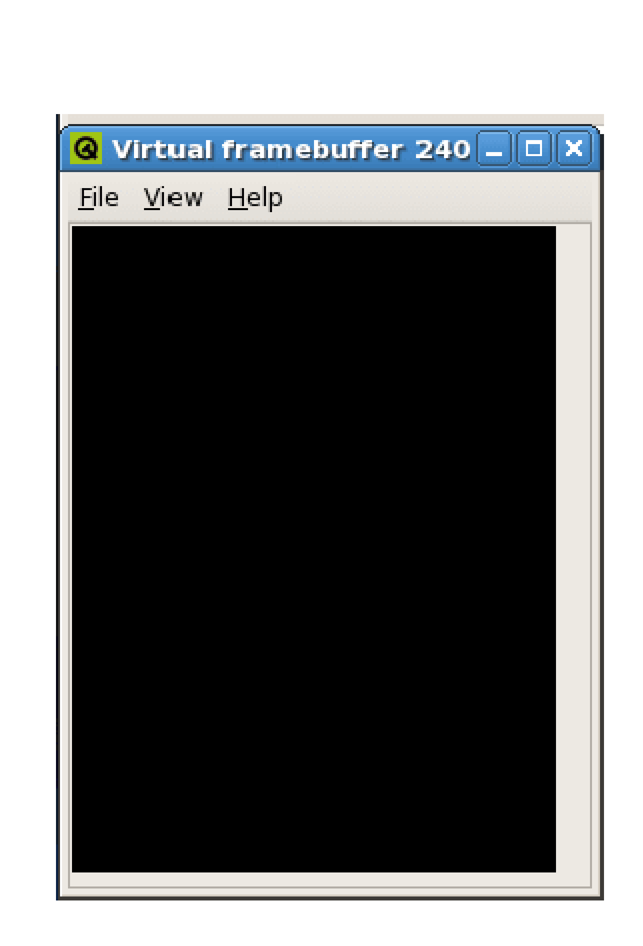



图 14.2　虚拟帧缓冲工具 qvfb

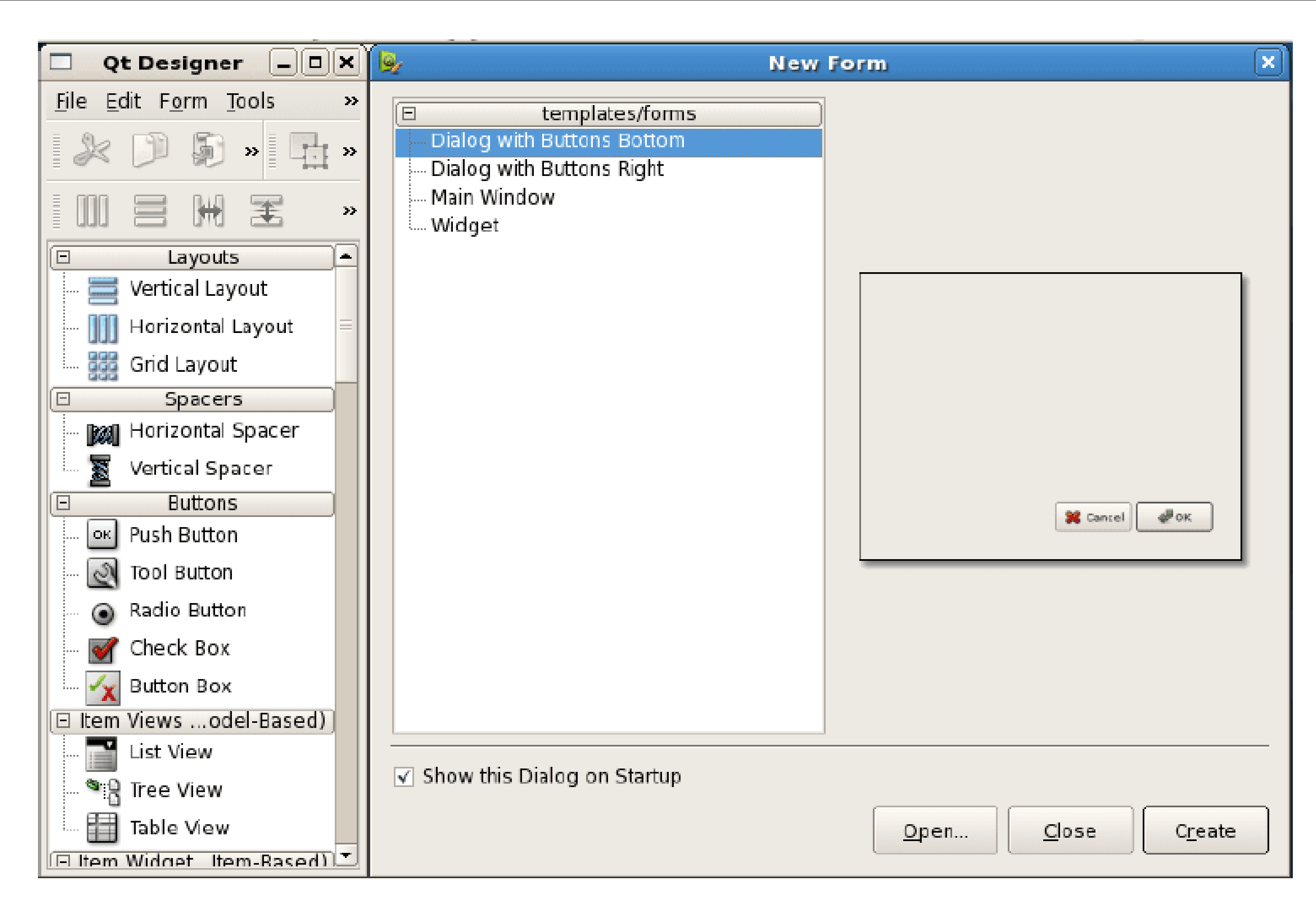



图 14.3　运行 Qt designer

在编译完 qt-x11-opensource-src-4.2.2 后，默认安装在/usr/local/Trolltech/Qt-4.2.2 路径下。将生成的工具复制到/usr/local/Trolltech/Qt-4.2.2/bin 目录下，同时删除 debug 文件，删除安装目录，修改指定工具的路径。

```
# cd /usr/local/arm/mini2440/qt-x11-opensource-src-4.2.2/bin/
# mv -f* /usr/local/Trolltech/Qt-4.2.2/bin/          //将工具复制到安装目录
# rm -rf qt-x11-opensource-src-4.2.2/                 //删除安装目录
# cd /usr/local/Trolltech/Qt-4.2.2/bin
# rm -f *.debug                                       //删除 debug 文件
# export QTDIR=/usr/local/Trolltech/Qt-4.2.2          //修改环境变量
# export PATH=$QTDIR/bin:$PATH
# export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

## 14.2.2　编译 Qtopia Core

首先解压在/usr/local/arm/mini2440 目录下，配置在 X86 平台上编译，编译过程如下：

```
#tar zxvf qtopia-core-opensource-src-4.3.5.tar.gz
# cd qtopia-core-opensource-src-4.3.5
#./configure  -embedded  x86  - depths  4,8,16,24,32  -qconfig  full  -qvfb
-qt-libjpeg -qt-libpng -qt-gif
#yes
```

**注意：** embedded 指定 CPU 的体系结构，在刚接触 Qtopia 时，建议先编译 x86 版本，并加入 qvfb 的支持，在主机上模拟帧缓冲运行 Qtopia Core 程序。熟悉了 Qtopia 的开发过程后，将其编译成 ARM 版本，再进行交叉编译和移植。

接下来进行编译和安装，编译和安装使用 gmake。

```
#gmake
#gmake install
```

安装完成后默认会安装在/usr/local/Trolltech/QtopiaCore-4.3.5 路径下。与 Qt 安装过程

类似，安装完成后也同样设置环境变量及删除安装文件和 debug 文件。

```
#cd $HOME
#vi .bashrc
```

在文件后面添加下面内容并保存。然后执行 source 命令使之生效。

```
export PATH=$PATH:/usr/local/Trolltech/QtopiaCore-4.3.5/bin
export QTDIR=/usr/local/Trolltech/QtopiaCore-4.3.5
export PATH=$QTDIR/bin:$PATH
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
# source .bashrc
```

### 14.2.3　Qtopia 在 X86 平台上的应用开发

开发 Qtopia 程序的主要部分包括类的定义、类的实现和显示 3 个部分。以 Dialog 为例，在设计 Qtopia 时，在开发初期在 Dialog.h 中定义好其包含的属性和功能，在详细设计时在 Dialog.cpp 中实现类方法，同时编写其测试代码即显示 main.cpp。Dialog.h 和 Dialog.cpp 作为一个类的完整描述，可以看成一个小模块。

#### 1. 类的设计

设置 Qt 类时，属性包括界面中包含的对象；方法包括创建或者删除这些对象的操作等。下面为 Dialog 类的设计，它继承 QDialog。继承 QDialog 的公共属性和方法，可省去重新设计的时间。

```
class Dialog : public QDialog
{
    Q_OBJECT

public:
    Dialog();

private:
    void createMenu();                          //创建菜单
    void createHorizontalGroupBox();            //创建水平组合框
    void createGridGroupBox();                  //创建网格组合框

    enum { NumGridRows = 3, NumButtons = 4 };

    QMenuBar *menuBar;                          //包括菜单
    QGroupBox *horizontalGroupBox;              //水平组合框
    QGroupBox *gridGroupBox;                    //网格组合框
    QTextEdit *smallEditor;                     //小文本编辑控件
    QTextEdit *bigEditor;                       //大文本编辑控件
    QLabel *labels[NumGridRows];                //标签
    QLineEdit *lineEdits[NumGridRows];          //输入框空间
    QPushButton *buttons[NumButtons];           //按钮
    QDialogButtonBox *buttonBox;                //对话框按钮

    QMenu *fileMenu;
    QAction *exitAction;
};
```

## 2．类的实现

类的实现，即在概要设计的基础上对需求的细化过程。实现在类设计阶段的细节，为类的方法填充实现部分。

```
/*构造函数，创建初始化对话框中的所有控件*/
Dialog::Dialog()
{
    createMenu();                               //调用创建菜单方法
    createHorizontalGroupBox();                 //调用创建水平组合框方法
    createGridGroupBox();                       //调用创建网格组合框方法

    bigEditor = new QTextEdit;                  //实例化一个文本编辑框
    bigEditor->setPlainText(tr("This widget takes up all the remaining
    space "
                          "in the top-level layout."));
                                                //设置文本编辑框的初始内容

    buttonBox = new QDialogButtonBox(QDialogButtonBox::Ok
                                | QDialogButtonBox::Cancel);
                                                //初始化一个对话框按钮
    connect(buttonBox, SIGNAL(accepted()), this, SLOT(accept()));
                                                //设置按钮的接收信号与槽
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
                                                //设置按钮的拒绝信号与槽

    QVBoxLayout *mainLayout = new QVBoxLayout;
                                                //实例化一个界面显示风格
    mainLayout->setMenuBar(menuBar);            //界面菜单设置为刚才实例的对象
    mainLayout->addWidget(horizontalGroupBox);          //添加水平组合框
    mainLayout->addWidget(gridGroupBox);        //添加网格组合框
    mainLayout->addWidget(bigEditor);           //添加文本编辑框
    mainLayout->addWidget(buttonBox);           //添加按钮
    setLayout(mainLayout);                      //设置显示风格

    setWindowTitle(tr("Basic Layouts"));        //设置主题
}

void Dialog::createMenu()                       //创建菜单方法
{
    menuBar = new QMenuBar;                     //实例化一个菜单栏对象

    fileMenu = new QMenu(tr("&File"), this);            //新建“文件”菜单
    exitAction = fileMenu->addAction(tr("E&xit"));
                                        //在“文件”菜单中增加一个“退出”方法
    menuBar->addMenu(fileMenu);

    connect(exitAction, SIGNAL(triggered()), this, SLOT(accept()));
                                        //连接“退出”操作的信号与槽
}

void Dialog::createHorizontalGroupBox()//创建水平组合框
{
    horizontalGroupBox = new QGroupBox(tr("Horizontal layout"));
    QHBoxLayout *layout = new QHBoxLayout;
```

```
    for (int i = 0; i < NumButtons; ++i) {
        buttons[i] = new QPushButton(tr("Button %1").arg(i + 1));
     layout->addWidget(buttons[i]);
    }
    horizontalGroupBox->setLayout(layout);
}

void Dialog::createGridGroupBox()              //创建网格组合框
{
    gridGroupBox = new QGroupBox(tr("Grid layout"));
    QGridLayout *layout = new QGridLayout;

    for (int i = 0; i < NumGridRows; ++i) {
     labels[i] = new QLabel(tr("Line %1:").arg(i + 1));
     lineEdits[i] = new QLineEdit;
     layout->addWidget(labels[i], i + 1, 0);
     layout->addWidget(lineEdits[i], i + 1, 1);
    }

    smallEditor = new QTextEdit;
    smallEditor->setPlainText(tr("This widget takes up about two thirds of
the "
                                "grid layout."));
    layout->addWidget(smallEditor, 0, 2, 4, 1);

    layout->setColumnStretch(1, 10);
    layout->setColumnStretch(2, 20);
    gridGroupBox->setLayout(layout);
}
```

### 3．实现测试文件

在详细设计完一个功能或者一个类后，就开始对该功能或类进行测试。下面就是对 Dialog 类的测试文件 main.cpp。

```
#include <QApplication>
#include "dialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Dialog dialog;
    return dialog.exec();
}
```

### 4．编写工程文件

在详细设计完成后，进入集成测试阶段时需要对多个模块进行继承测试。这时需要编写工程文件，在其他项目中可能是编写整个工程的 Makefile 文件。在 Qtopia 程序中，qmake 会辅助生成 Makefile 文件，所以只需要编写工程文件 dialog.pro。

```
HEADERS     = dialog.h
SOURCES     = dialog.cpp \
              main.cpp

target.path = $$[QT_INSTALL_EXAMPLES]/layouts/basiclayouts
sources.files = $$SOURCES $$HEADERS *.pro
```

```
sources.path = $$[QT_INSTALL_EXAMPLES]/layouts/basiclayouts
INSTALLS += target sources
```

### 5．生成Makefile、编译和运行

这部分包括通过工程文件生成 Makefile 和编译工程生成可执行文件，最后运行结果，测试结果是否与概要设计相符合。

```
#qmake -o Makefile dialog.pro
#make
```

打开两个终端，一个运行虚拟帧缓冲 avfb，另一个运行 Qtopia 应用程序。运行效果如图 14.4 所示。

```
#qvfb
#./dialog -qws
```

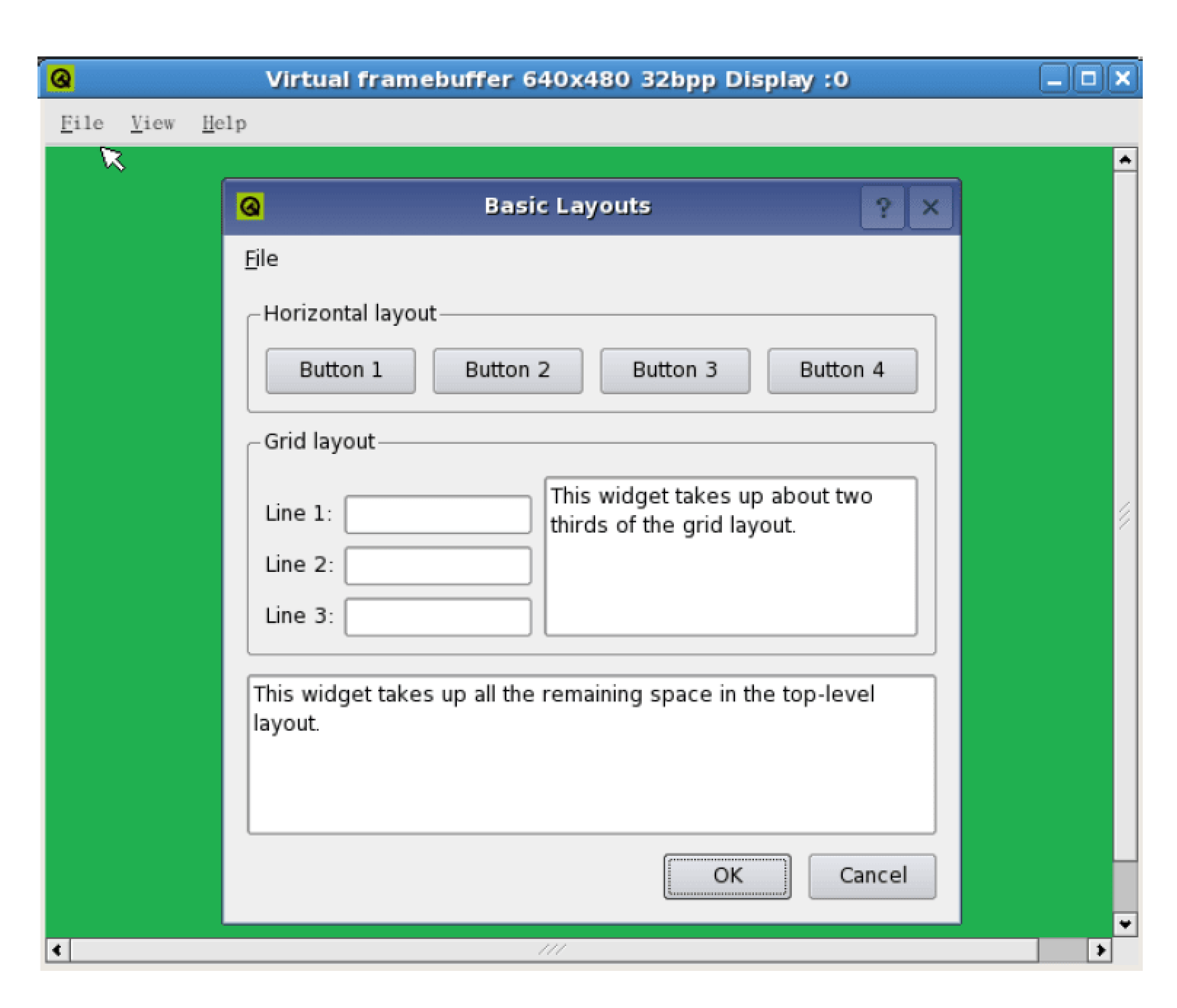



图 14.4　运行结果

**注意：**运行 avfb 后在虚拟帧缓冲对话框出现后，选择 configure 命令，在对话框中选择显示的大小为 640 × 480。运行应用程序时带参数-qws。

## 14.3　Qtopia Core 在嵌入式 Linux 上的移植

Qtopia Core 的前身是 Qt/Embedded，继承了 Qt 4 的功能与优点，拥有与桌面系统的 Qt 相同的应用程序编程接口（API）和工具包。Qtopia Core 是一个为嵌入式设备上的图形用户接口和应用开发而订做的 C++工具开发包。Qtopia Core 采用与桌面版本同样的 API，

但在其内部实现上做了很多调整以适应硬件限制的嵌入式环境。Qtopia Core 包含多个 Qt 工具，可以进行快速优化和开发。

## 14.3.1　Qtopia Core 移植准备

交叉编译 Qtopia Core 时需要准备两个程序：

- qt-x11-free-3.3.8b.tar.gz，编译其目的主要是为了获得库 libqjpeg。
- qtopia-core-opensource-src-4.3.5.tar.gz，编译、安装获得环境 QtopiaCore-4.3.5-arm。

**注意：** qt-x11-free-3.3.8b.tar.gz 不是必须的，因为在编译 qtopia-core-opensource-src-4.3.5 时出现缺少库 libqjpeg.so 支持的错误，而刚好编译 qt-x11-free-3.3.8b 后 qt-x11-free-3.3.8b/plugins/imageformats 目录下存在文件 libqjpeg.so。

将 qt-x11-free-3.3.8b.tar.gz 和 qtopia-core-opensource-src-4.3.5.tar.gz 赋值到/usr/local/arm/mini2440 目录下。14.1 节中已经介绍过 qt-x11-free-3.3.8b 编译方法了，这里直接给出其编译 qt-x11-free-3.3.8b 的命令。不对命令进行详细解释，有疑问可以参考 14.1.1 节。

```
#tar zxvf qt-x11-free-3.3.8b.tar.gz
#cd qt-x11-free-3.3.8b
#./configure
#make
```

如果前面读者已经编译过 qt-x11-free-3.3.8b ，而且确定在 qt-x11-free-3.3.8b/plugins/imageformats 目录下存在文件 libqjpeg.so，可以不必重新编译 qt-x11-free-3.3.8b。

## 14.3.2　交叉编译 Qtopia Core

交叉编译 Qtopia Core 的过程类似其他程序的交叉编译方法，总体包括 4 步：设置交叉编译器、配置生成 Makefile 文件、编译和安装。下面详细介绍整个交叉编译过程和注意细节。

### 1．设置交叉编译器

设置交叉编译器之前查看本机的交叉编译工具。如果读者现在还没有安装交叉编译工具，请查看前面的章节安装交叉编译工具。本机使用的交叉编译工具为 arm-linux-gcc-4.3.2。使用命令 arm-linux-gcc –v 查看本机安装好的交叉编译工具。

```
# arm-linux-gcc -v
Using built-in specs.
Target: arm-none-linux-gnueabi                    //本机的交叉编译器
```

进入 mini2440 目录，解压 qtopia-core-opensource-src-4.3.5.tar.gz。修改 mkspecs/qws/linux-arm-g++下的 qmake.conf 文件，把文件里面的编译器指定为 arm-none-linux-gnueabi 用 arm-none-linux-gnueabi-gcc 和 arm-none-linux-gnueabi-g++替代以下的 arm-linux-gcc 和 arm-linux-g++。具体的执行过程如下：

```
# tar zxvf qtopia-core-opensource-src-4.3.5.tar.gz
```

```
# cd qtopia-core-opensource-src-4.3.5
# cd mkspecs/qws/linux-arm-g++/
# vi qmake.conf
```

在文件 qmake.conf 中将与编译器有关的内容修改如下：

```
QMAKE_CC                  = arm-linux-gcc
QMAKE_CXX                 = arm-linux-g++
QMAKE_LINK                = arm-linux-g++
QMAKE_LINK_SHLIB          = arm-linux-g++
```

改为

```
QMAKE_CC                  = arm-none-linux-gnueabi-gcc
QMAKE_CXX                 = arm-none-linux-gnueabi-g++
QMAKE_LINK                = arm-none-linux-gnueabi-g++
QMAKE_LINK_SHLIB          = arm-none-linux-gnueabi-g++
```

### 2．配置编译选项

进入 qtopia-core-opensource-src-4.3.5 目录下，使用 configure 命令创建 qmake 生成 Makefile 文件。

```
# cd /usr/local/arm/mini2440/qtopia-core-opensource-src-4.3.5/
#./configure -no-largefile -no-qt3support -nomake tools -make examples
-silent -xplatform qws/linux-arm-g++ -embedded arm -depths 16,18,24,32
-qt-kbd-tty -qt-kbd-usb -system-libjpeg -qt-gfx-transformed -confirm-
license
配置完成后生成 Makefile、plugins 和 qmake 等目录。
```

### 3．添加qjpeg库支持

为提供 qjpeg 库支持，将 qt-x11-free-3.3.8b/plugins/imageformats 目录下的文件 libqjpeg.so，复制到目录/usr/local/arm/mini2440/qtopia-core-opensource-src-4.3.5/plugins/imageformats/下。

```
#cp /usr/local/arm/mini2440/qt-x11-free-3.3.8b/plugins/imageformats/
libqjpeg.so\
/usr/local/arm/mini2440/qtopia-core-opensource-src-4.3.5/plugins/imagef
ormats/
```

### 4．修改文件qdrawhelper.cpp

如果读者使用的是 arm-linux-gcc-3.3.2 版本的交叉编译器，或许不需要做本步的修改。本机使用 arm-linux-gcc-4.3.2 进行编译的时候出现错误提示 explicit template specialization cannot have a storage class。解决的方法做以下修改：

```
#vi src/gui/painting/qdrawhelper.cpp        //去掉该文件两处 static
```

第 1 处：（根据报错的行数为 5905）

```
template <>
static inline void qt_memrotate90_template<quint18, quint32>(const quint32
*src,
                              int srcWidth, int srcHeight, int srcStride,
```

```
                        quint18 *dest, int dstStride)
```

改为：

```
template <>
inline void qt_memrotate90_template<quint18, quint32>(const quint32 *src,
                                   int srcWidth, int srcHeight,int srcStride,
                                   quint18 *dest, int dstStride)
```

第 2 处：（根据报错的行数为 5929）

```
template <>
static inline void qt memrotate90 template<quint24, quint32>(const quint32
*src,
                                   int srcWidth, int
                                   srcHeight, int srcStride,
                                   quint24 *dest, int dstStride)
```

改为：

```
template <>
inline void qt_memrotate90_template<quint24, quint32>(const quint32 *src,
                     int srcWidth, int srcHeight, int srcStride,
                     quint24 *dest, int dstStride)
```

在没有进行修改前编译会出现下面的错误。解决错误的方法有：根据错误的内容修改代码，或者安装低版本的编译器。

```
painting/qdrawhelper.cpp:5905: error: explicit template specialization
cannot have a storage class
painting/qdrawhelper.cpp:5929: error: explicit template specialization
cannot have a storage class
make[1]: *** [.obj/release-shared-emb-arm/qdrawhelper.o] 错误 1
make[1]: Leaving directory `/usr/local/arm/mini2440/qtopia-core-
opensource-src-4.3.5/src/gui'
make: *** [sub-gui-make_default-ordered] 错误 2
```

### 5．编译、安装Qtopia Core

执行 make 和 make install，将 QtopiaCore-4.3.5-arm 安装到默认的路径/usr/local/Trolltech 下。

```
#make
#make install
```

安装完成后，检查在/usr/local 目录下多了个目录 Trolltech。在 Trolltech 下多了目录 QtopiaCore-4.3.5-arm。/usr/local/Trolltech/QtopiaCore-4.3.5-arm 就是 qtopia-core-opensource-src-4.3.5 交叉编译、安装后的默认路径。

### 6．修改环境变量PATH

修改环境变量 PATH，添加路径/usr/local/Trolltech/QtopiaCore-4.3.5-arm/bin，使得系统能够直接找到工具 moc、qmake、rcc 和 uic。

```
#cd $HOME
#vi .bashrc
```

```
在末尾添加
export PATH=$PATH:/usr/local/Trolltech/QtopiaCore-4.3.5-arm/bin
```

## 14.3.3　编译内核

因为在编译 Qtopia Core 时，编译器为 arm-none-linux-gnueabi。因此内核也应该采用此编译器进行编译，并且内核配置也要对 ARM EABI 进行支持。执行 make menuconfig 对内核进行配置，添加对 EABI 的支持。进入内核配置界面后选择 Kernel Features --->进入 Kernel Features 配置界面，勾选 EABI 项，如图 14.5 所示。

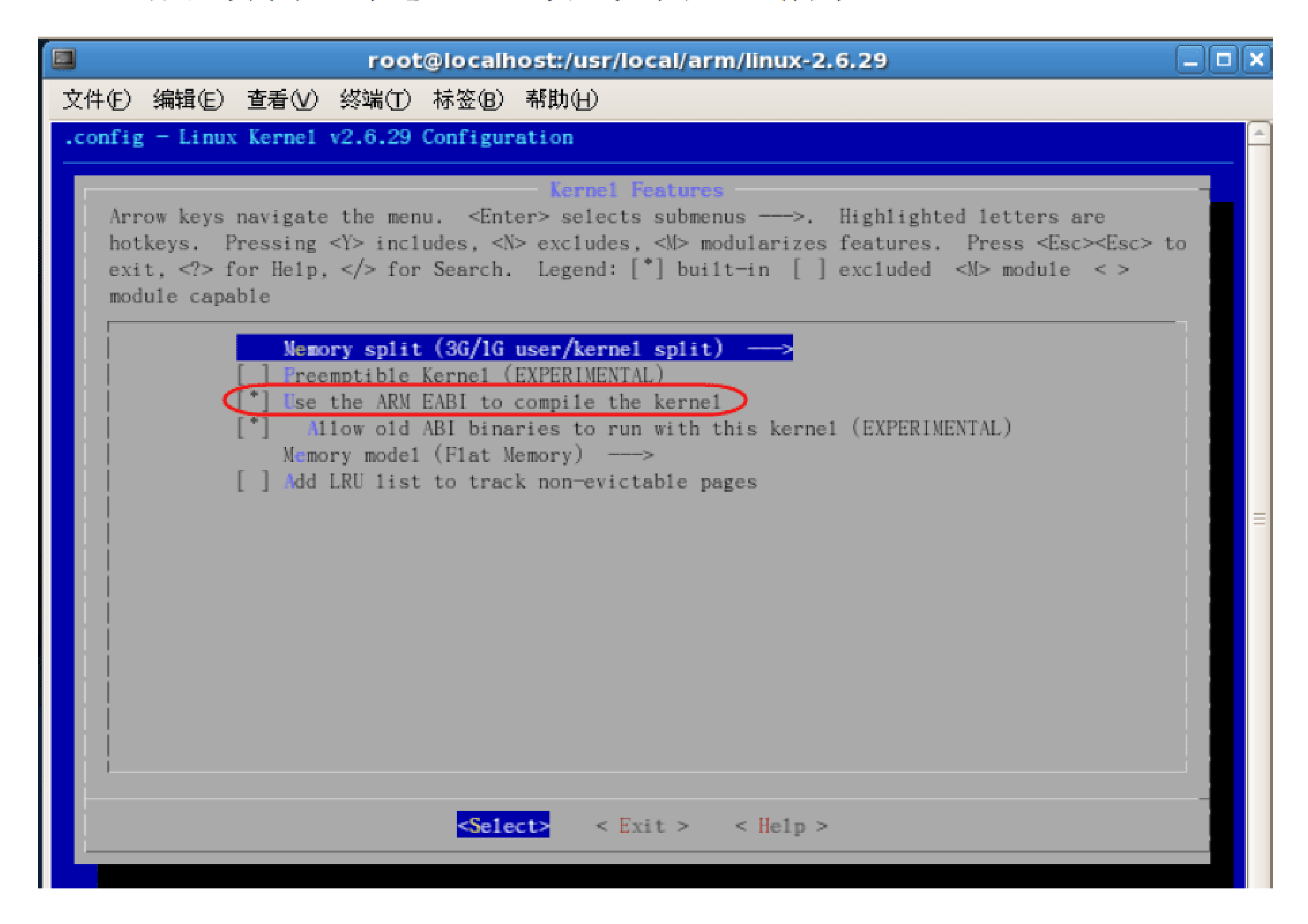



图 14.5　配置内核对 EABI 的支持

在内核源码目录下，修改内核 Makefile，将编译器设置为 arm-none-linux-gnueabi-。

```
#vi Makefile
ARCH            = arm
CROSS_COMPILE   = arm-none-linux-gnueabi-
```

执行清理并重新生成内核映像文件。

```
#make clean
#make zImage
```

生成新的映像文件后，重新烧写内核和文件系统。

## 14.3.4　应用程序开发

应用程序的开发往往是项目的核心内容，是系统功能需求部分的重点。应用程序的开发应该实现项目的需求或者合同中指定的功能，或者实现子系统或模块的功能。界面程序有针对普通用户和管理员，在具体项目中应根据实际情况进行功能设计。本例引用例子程

序 standarddialogs 介绍 GUI 的移植过程。

### 1．编写应用程序

以安装路径下的例子介绍，进入下面的目录可以查看是否存在.cpp 文件和.h 文件/usr/local/Trolltech/QtopiaCore-4.3.5-arm/examples/dialogs/standarddialogs。编写 dialog.h 文件定义类，包括属性和方法。

```
class Dialog : public Qdialog      //类的名字和继承关系
{
    Q_OBJECT              //使用信号与槽机制时，类的声明中必须加上 Q_OBJECT 语句
public:
    Dialog(QWidget *parent = 0);

private slots:                     //类的方法定义
    void setInteger();
    void setDouble();
    void setItem();
    void setText();
    void setColor();
    void setFont();
    void setExistingDirectory();
    void setOpenFileName();
    void setOpenFileNames();
    void setSaveFileName();
    void criticalMessage();
    void informationMessage();
    void questionMessage();
    void warningMessage();
    void errorMessage();

private:                           //类的属性定义
    QCheckBox *native;
    QLabel *integerLabel;
    QLabel *doubleLabel;
    QLabel *itemLabel;
    QLabel *textLabel;
    QLabel *colorLabel;
    QLabel *fontLabel;
    QLabel *directoryLabel;
    QLabel *openFileNameLabel;
    QLabel *openFileNamesLabel;
    QLabel *saveFileNameLabel;
    QLabel *criticalLabel;
    QLabel *informationLabel;
    QLabel *questionLabel;
    QLabel *warningLabel;
    QLabel *errorLabel;
    QErrorMessage *errorMessageDialog;

    QString openFilesPath;
};
#endif
```

编写类的实现，在文件 dialog.cpp 中实现类的方法。

```
Dialog::Dialog(QWidget *parent)
```

```
    : QDialog(parent)
{
    //构造一个 QErrorMessage 类型的对象赋给 errorMessageDialog
errorMessageDialog = new QErrorMessage(this);
    int frameStyle = QFrame::Sunken | QFrame::Panel;    //设置帧格式

    integerLabel = new QLabel;          //integerLabel 指向一个 QLabel 对象
    integerLabel->setFrameStyle(frameStyle);    //设置 integerLabel 帧格式
    QPushButton *integerButton = // integerButton 指向 QPushButton 类型实例
            new QPushButton(tr("QInputDialog::get&Integer()"));
//连接信号与槽，当 integerButton 被按下时，执行方法 setInteger()
connect(integerButton, SIGNAL(clicked()), this, SLOT(setInteger()));
//下面为设置空间的排列风格，其他的控件可以参照 integerButton 进行设置
QGridLayout *layout = new QGridLayout;
    layout->setColumnStretch(1, 1);
    layout->setColumnMinimumWidth(1, 250);
    layout->addWidget(integerButton, 0, 0);
    setLayout(layout);

    setWindowTitle(tr("Standard Dialogs"));
}
//当 integerButton 被按下时，该方法被调用
void Dialog::setInteger()
{
    bool ok;
    int i = QInputDialog::getInteger(this, tr("QInputDialog::
    getInteger()"),
                                 tr("Percentage:"), 25, 0, 100, 1, &ok);
    if (ok)
        integerLabel->setText(tr("%1%").arg(i));
}
```

### 2．编写工程文件standarddialogs.pro

编写工程文件包括指定依赖的头文件和包含的源文件。指定目标文件路径、源文件和源文件路径。

```
HEADERS       = dialog.h
SOURCES       = dialog.cpp \
                main.cpp
# install
target.path = $$[QT INSTALL EXAMPLES]/dialogs/standarddialogs
sources.files = $$SOURCES $$HEADERS *.pro
sources.path = $$[QT_INSTALL_EXAMPLES]/dialogs/standarddialogs
INSTALLS += target sources
```

### 3．利用qmake和工程文件standarddialogs.pro生成Makefile，并编译生成可执行文件

```
#qmake -o Makefile standarddialogs.pro
#make                          //执行 make 后生成可执行文件
```

生成的 Makefile 如下，是否与预期的编译器、库文件路径、头文件路径、链接库、使用的 qmake 等一致。

```
CC            = arm-none-linux-gnueabi-gcc
CXX           = arm-none-linux-gnueabi-g++
```

```
DEFINES       = -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_NETWORK_LIB -DQT_CORE_LIB
-DQT_SHARED
CFLAGS        = -pipe -O2 -Wall -W -D_REENTRANT $(DEFINES)
CXXFLAGS      = -pipe -O2 -Wall -W -D_REENTRANT $(DEFINES)
INCPATH          =  -I../../../mkspecs/qws/linux-arm-g++  -I.  -I../../../
include/QtCore   -I../../../include/QtCore   -I../../../include/QtNetwork
-I../../../include/QtNetwork -I../../../include/QtGui -I../../../include/
QtGui -I../../../include -I. -I.
LINK         = arm-none-linux-gnueabi-g++
LFLAGS        = -Wl,-rpath,/usr/local/Trolltech/QtopiaCore-4.3.5-arm/lib
LIBS                  =  $(SUBLIBS)    -L/usr/local/Trolltech/QtopiaCore-
4.3.5-arm/lib   -lQtGui   -L/usr/local/Trolltech/QtopiaCore-4.3.5-arm/lib
-lQtNetwork  -lQtCore -lm -lrt -ldl -lpthread
AR           = arm-none-linux-gnueabi-ar cqs
RANLIB        = arm-none-linux-gnueabi-ranlib           //编译器 lib 库
QMAKE   = /usr/local/Trolltech/QtopiaCore-4.3.5-arm/bin/qmake //qmake 路径
```

### 14.3.5　应用程序移植

在移植应用程序前，确定已经正确移植了所需的内核和文件系统，内核需要支持 ARM EABI 编译。接下来主要进行 3 部分工作，移植库、设置环境变量和移植应用程序。

#### 1．移植库文件到开发板

在开发板/usr/local 目录下新建目录 QtopiaCore，将上位机/usr/local/Trolltech /QtopiaCore-4.3.5-arm/lib 目录下的库文件复制到 QtopiaCore/lib 目录下。

```
libQtCore.so.4
libQtNetwork.so.4
libQtGui.so.4
```

#### 2．设置环境变量

```
export QTDIR=/usr/local/QtopiaCore
export QPEDIR=/usr/local/QtopiaCore
export LD_LIBRARY_PATH=$QTDIR/lib:/usr/local/lib:$LD_LIBRARY_PATH
```

#### 3．移植应用程序到开发板

将编译好的应用程序复制到/usr/local/ QtopiaCore 目录下，在该目录下执行./configdialog –qws &，运行程序。

```
./configdialog -qws &
```

## 14.4　小　　结

Qtopia 在嵌入式 GUI 应用中占了很大部分，现在其主要运用为电话和 PDA。对于使用 Qtopia 开发，读者需要深入学习 Qt 的 API 接口的使用，了解各个类的功能。Qtopia 安装、编译和移植只是带读者入门，为了完成项目，还需要更深入地学习 Qt 编程。

# 第 15 章　嵌入式数据库 Berkeley DB 移植

Berkeley DB 是一款健壮、高速的工业级嵌入式数据库产品。很多知名公司都采用了这款嵌入式数据库。Berkeley DB 的一个很重要的特点就是高速存储。在高流量、高并发的情况下，Berkeley DB 要比非嵌入式的数据库表现得更加出色。另外，它可以应用在各种操作系统上，也能编译成多种语言的 API 接口。本章将主要介绍在 Linux 上的编译，编译的接口为 C 和 C++。

## 15.1　数据库的基本概念

在移植和使用 Berkeley DB 之前，先了解数据库的一些基本概念。如果读者从来没有数据库设计的经验，或者从来没有使用过 DB2、SQL Server、MySQL、Oracle 等这些数据的经验，那么可以把数据库理解为一个存放数据的仓库。下面将介绍以何种方式存放数据到仓库、以何种方式从仓库中取出数据以及提高这些方法的效率。任何数据库的设计都是围绕这些主题来实现的，当然嵌入式数据库除了考虑效率、安全性外，还需考虑占用的空间。

### 15.1.1　利用文档和源代码

首先读者应该从官方网站下载 Berkeley DB 的源码包。建议先在 Windows 目录下解压后，进入 docs 目录下。如果读者习惯使用 API 文档，会知道双击 index.html 就能进入帮助文档主页。下面有关数据库基本方法的介绍都是参照该 API 文档进行分析。同时，读者还可以借助一些工具（如 Source Insight 或者 UltraEdit）打开源代码，使用工具的同步文件功能。在完成上述两项事情后，会很大程度提高读者的学习和开发进度。

### 15.1.2　创建环境句柄

在 Berkeley DB 中，创建一个所有应用程序存放数据的环境。在代码中，将通过一个环境句柄来引用环境，该句柄类型为 DB_ENV。读者将使用这一句柄操作此环境。利用 API 文档，找到 Programmatic APIs，先查看提供给 C 语言的 API。

在 C 中使用该 API：

```
#include <db.h>
Int db_env_create(DB_ENV **dbenvp, u_int32_t flags);
```

功能描述：函数 db_env_create()为 Berkeley DB 环境创建 DB_ENV 结构体句柄。为该结构体分配内存，并返回指向结构体的指针 dbenvp。函数 db_env_create()返回 0 表示创建

成功，非 0 表示创建失败。

在 C++中使用该 API：

```
#include <db cxx.h>
class DbEnv {
public:
        DbEnv(u_int32 flags);
        ~DbEnv();

        DB_ENV *DbEnv::get_DB_ENV();
        const DB_ENV *DbEnv::get_const_DB_ENV() const;
        static DbEnv *DbEnv::get_DbEnv(DB_ENV *dbenv);
        static const DbEnv *DbEnv::get_const_DbEnv(const DB_ENV *dbenv);
        ...
};
```

功能描述：DbEnv 对象是 Berkeley DB 环境的句柄。该构造函数创建 DbEnv 对象，并为该对象分配空间。在调用 DbEnv::close() 或 DbEnv::remove()方法时释放该空间。

**注意：** 在这里结合 Berkeley DB 数据库讲解数据库的基本概念时，如果读者并没有编译、安装 Berkeley DB 就使用上面介绍的 API 或者 API 文档中的 API 接口函数，将会在程序编译时不能通过。在后面将会详细介绍如何安装成 C 或 C++的库。如果读者热切希望能先试验一番，可以先看如何编译和安装部分，然后回头看基本概念部分。

### 15.1.3　创建数据库句柄

Berkeley DB 创建一个数据库句柄来代表创建的表。在 C 中使用该 API：

```
#include <db.h>
int db_create(DB **dbp, DB_ENV *dbenv, u_int32_t flags);
```

功能描述：函数 db_create()为 Berkeley DB 数据库创建一个结构为 DB 的句柄。为该结构体分配内存，并返回指向结构体的指针 dbp。

在 C++中使用该 API：

```
#include <db cxx.h>
class Db {
public:
        Db(DbEnv *dbenv, u_int32_t flags);
        ~Db();

         DB *Db::get_DB();
         const DB *Db::get_const_DB() const;
         static Db *Db::get_Db(DB *db);
         static const Db *Db::get_const_Db(const DB *db);
         ...
};
```

功能描述：该构造函数为 Berkeley DB 数据库创建一个 DB 对象的句柄。该构造函数为对象分配空间，在调用 Db::close()、Db::remove()或 Db::rename()时释放该空间。

## 15.1.4　打开数据库

打开由文件和数据库参数表示的数据库，为读写数据库做准备。当前支持 Berkeley DB 数据库文件格式有 Btree、Hash、Queue 和 Recno。Btree 格式表示的是一个有序的平衡树结构；Hash 格式表示的是一个动态、可扩展的散列图；Queue 格式支持快速或通过逻辑记录号访问固定长度的连续记录；Recno 格式支持访问固定或可变长度记录，支持连续访问或者通过逻辑号访问，并且可以通过文本任意备份。

在 C 中使用该 API：

```
#include <db.h>
int
DB->open(DB *db, DB_TXN *txnid, const char *file,
   const char *database, DBTYPE type, u_int32_t flags, int mode)
```

功能描述：函数 DB->open()打开失败时返回一个非 0 错误值，成功返回 0。如果函数 DB->open()失败，函数 DB->close()被调用丢弃 DB 句柄。

在 C++中使用该 API：

```
#include <db_cxx.h>

int
Db::open(DbTxn *txnid, const char *file,
const char *database, DBTYPE type, u_int32_t flags, int mode);
```

功能描述：函数 Db::open()打开通过文件或数据库表示的数据库。函数 DB->open()打开失败时返回一个非 0 错误值，成功返回 0。如果函数 DB->open()失败，函数 DB->close()被调用丢弃 DB 句柄。

## 15.1.5　DBT 结构

在 Berkeley DB 中，DBT 结构用来定义 Key/Value 对。

在 C 中使用该结构：

```
#include <db.h>
   typedef struct {
      void *data;
      u_int32_t size;
      u_int32_t ulen;
      u_int32_t dlen;
      u_int32_t doff;
      u_int32_t flags;
   } DBT;
```

功能描述：DBT 结构的所有域在第一次使用前没有明确地初始化为空字节。在声明该结构体时应该使用库函数 memset()进行初始化。默认的情况下，flags 被设置为 0。

在 C++中使用该结构：

```
#include <db_cxx.h>
class Dbt {
public:
```

```
        Dbt(void *data, size_t size);
        Dbt();
        Dbt(const Dbt &);
        Dbt &operator = (const Dbt &);
        ~Dbt();

        void *get_data() const;
        void set_data(void *);

        u_int32_t get_size() const;
        void set_size(u_int32_t);

        u_int32_t get_ulen() const;
        void set_ulen(u_int32_t);

        u_int32_t get_dlen() const;
        void set_dlen(u_int32_t);

        u_int32_t get_doff() const;   //返回局部记录的偏移（bytes）
        void set_doff(u_int32_t);//设置在被应用程序读写时，局部记录的偏移（bytes）

        u_int32_t get_flags() const;
        void set_flags(u_int32_t);

        DBT *Dbt::get_DBT();
        const DBT *Dbt::get_const_DBT() const;
        static Dbt *Dbt::get_Dbt(DBT *dbt);
        static const Dbt *Dbt::get_const_Dbt(const DBT *dbt);
};
```

功能描述：Berkeley DB 数据库中，类 Dbt 用于将 key 和 data 项进行编码。key 和 data 项都被表示为 Dbt 对象。

## 15.1.6　存取数据

数据库的存取操作是数据库的主要操作，在 Berkeley DB 数据库中存取操作主要是对 key/data 进行存取。

在 C 中使用该 API：

```
#include <db.h>
int DB->put(DB *db,
    DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);
```

功能描述：函数 DB->put()存放 key/data 到数据库 DB 中。

```
int DB->get(DB *db, DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);
int DB->pget(DB *db, DB_TXN *txnid, DBT *key, DBT *pkey, DBT *data, u_int32_t
flags);
```

功能描述：函数 DB->get()和 DB->pget()从数据库 DB 中取 key/data。

在 C++中使用该 API：

```
#include <db_cxx.h>
int Db::put(DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags);
```

功能描述：函数 DB->put()存放 key/data 到数据库 DB 中。

```
int Db::get(DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags);
int Db::pget(DbTxn *txnid, Dbt *key, Dbt *pkey, Dbt *data, u_int32_t flags);
```

功能描述：函数 Db::get ()和 Db::pget ()从数据库 DB 中取 key/data。

### 15.1.7　关闭数据库

Berkeley DB 数据库使用完后，应该关闭数据库，释放打开的资源。

在 C 中使用该 API：

```
#include <db.h>
int DB->close(DB *db, u_int32_t flags);
```

功能描述：函数 DB->close() flushes 将所有数据库信息缓冲到磁盘，关闭任何打开游标，释放分配的任何资源，关闭任何文件。

在 C++中使用该 API：

```
#include <db cxx.h>
int Db::close(u_int32_t flags);
```

功能描述：函数 DB->close() flushes 任何缓存数据库信息到磁盘，关闭任何打开游标，释放分配的任何资源，关闭任何文件。

本节以 Berkeley DB 数据库为例介绍了数据库有关操作的基本概念。有关数据库的其他概念还有事务处理、互斥锁、内存池和操作异常等。这些概念都可以在 Berkeley DB 的 API 文档上找到，读者可以自己查阅该文档。

## 15.2　Berkeley DB 数据库安装

Berkeley DB 数据库不仅可以针对不同的系统安装，还可以针对不同的语言安装。安装成 C 库，使用为 C 提供的 API 就可以使用该数据库。也可以被安装成 C++库或 Java 库，使用为 C++或 Java 提供的 API 就可以使用该数据库。下面主要介绍安装为安装成 C 库、C++库和交叉编译安装过程。

### 15.2.1　安装成 C 库

在安装 Berkeley DB 数据库前，读者可以去 Berkeley DB 官方网站下载最新版的 Berkeley DB 数据库源代码。在写作本书时，Berkeley DB 的最新版本为 db-4.8.26.tar.gz。

（1）复制 db-4.8.26.tar.gz 到/usr/local 目录下，然后解压 db-4.8.26.tar.gz 到/usr/local。

```
# tar zxvf db-4.8.26.tar.gz
#cd db-4.8.26
```

（2）新建一个编译目录 build_linux，进入该目录。使用 configure 命令，按默认方式进行编译安装，就可以安装成 C 库。

```
# mkdir build_linux                //建立编译目录
# cd build linux
# ../dist/configure                //生成 Makefile，默认生成 C 库
# make                             //编译
# make install                     //安装
```

（3）正确编译、安装完成后，默认的安装路径为/usr/local/Berkeley DB.4.8。在该路径下生成目录 bin、docs、include 和 lib。在 lib 路径下生成的库文件有 libdb-4.8.a、libdb-4.8.la、libdb-4.8.so、libdb-4.so、libdb.a 和 libdb.so。

如果要卸载 Berkeley DB，可以在 build_linux 目录下使用 make uninstall 命令进行卸载。

```
# cd /usr/local/db-4.8.26/build linux
# make uninstall
```

卸载完成后，/usr/local/Berkeley DB.4.8 目录下的所有子目录均为空。

## 15.2.2　安装成 C++库

安装成 C++库与安装成 C 库基本类似。编译时唯一的区别在于 configure 时带上参数--enable-cxx。下面给出编译成 C++库的命令过程：

```
#cd /usr/local/db-4.8.26/build_linux   /*如果用户没有建立目录 build_linux，需
                                         在此步前执行#mkdir /usr/local/ db-
                                         4.8.26/build_linux，建立编译目录*/
# ../dist/configure --enable-cxx       //编译成 C++库
# make                                 //编译
# make install                         //安装
```

**注意：** *在此步编译的过程中，可以看到执行编译时采用的编译器为 g++而不是 GCC 了。*

安装完成后，进入/usr/local/Berkeley DB.4.8/lib 目录查看生成的库文件，与编译成 C 库有何区别。

```
libdb-4.8.a   libdb-4.so        libdb cxx-4.8.la   libdb cxx.a
libdb-4.8.la  libdb.a           libdb_cxx-4.8.so   libdb_cxx.so
libdb-4.8.so  libdb_cxx-4.8.a   libdb_cxx-4.so     libdb.so
```

安装结束后，清除编译/usr/local/db-4.8.26/build_linux 目录下的所有文件。

## 15.2.3　交叉编译安装 Berkeley DB

交叉编译安装 Berkeley DB 时，以安装成 C++库为例。安装成 arm-linux 环境下的库，两点主要的区别是编译器选择 arm-linux-g++，平台选择 ARM。在配置 configure 参数时，如果忘记了该参数，可以使用-help 进行查看。

```
# mkdir /usr/local/db-4.8.26/build_arm_linux
#cd /usr/local/db-4.8.26/build_arm_linux
#../dist/configure -help
```

查看默认参数配置细节。编译或者交叉编译其他源代码也是对类似的参数进行配置，然后进行编译和安装。

```
Fine tuning of the installation directories:
  --bindir=DIR           user executables [EPREFIX/bin]
  --sbindir=DIR          system admin executables [EPREFIX/sbin]
  --libexecdir=DIR       program executables [EPREFIX/libexec]
  --sysconfdir=DIR       read-only single-machine data [PREFIX/etc]
  --sharedstatedir=DIR   modifiable architecture-independent data
                         [PREFIX/com]
  --localstatedir=DIR    modifiable single-machine data [PREFIX/var]
  --libdir=DIR           object code libraries [EPREFIX/lib]
  --includedir=DIR       C header files [PREFIX/include]
  --oldincludedir=DIR    C header files for non-gcc [/usr/include]
  --datarootdir=DIR      read-only arch.-independent data root
                         [PREFIX/share]
  --datadir=DIR          read-only architecture-independent data
                         [DATAROOTDIR]
  --infodir=DIR          info documentation [DATAROOTDIR/info]
  --localedir=DIR        locale-dependent data [DATAROOTDIR/locale]
  --mandir=DIR           man documentation [DATAROOTDIR/man]
  --docdir=DIR           documentation root
                       [DATAROOTDIR/doc/db-4.8.26]
  --htmldir=DIR          html documentation [DOCDIR]
  --dvidir=DIR           dvi documentation [DOCDIR]
  --pdfdir=DIR           pdf documentation [DOCDIR]
  --psdir=DIR            ps documentation [DOCDIR]

Program names:
  --program-prefix=PREFIX        prepend PREFIX to installed program names
  --program-suffix=SUFFIX        append SUFFIX to installed program names
  --program-transform-name=PROGRAM   run sed PROGRAM on installed program
                                     names

System types:      //交叉编译需要设置的地方
  --build=BUILD     configure for building on BUILD [guessed]
  --host=HOST       cross-compile to build programs to run on HOST [BUILD]

Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-
                          FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --disable-bigfile       Obsolete; use --disable-largefile instead.
  --disable-cryptography  Do not build database cryptography support.
  --disable-hash          Do not build Hash access method.
  --disable-partition     Do not build partitioned database support.
  --disable-compression   Do not build compression support.
  --disable-mutexsupport  Do not build any mutex support.
  --disable-atomicsupport Do not build any native atomic operation support.
  --disable-queue         Do not build Queue access method.
  --disable-replication   Do not build database replication support.
  --disable-statistics    Do not build statistics support.
  --disable-verify        Do not build database verification support.
  --enable-compat185      Build DB 1.85 compatibility API.
  --enable-cxx            Build C++ API.   (编译成 C++库时设置)
  --enable-debug          Build a debugging version.
  --enable-debug_rop      Build a version that logs read operations.
  --enable-debug_wop      Build a version that logs write operations.
  --enable-diagnostic     Build a version with run-time diagnostics.
  --enable-dump185        Build db_dump185(1) to dump 1.85 databases.
  --enable-java           Build Java API.   (编译成 Java 库时设置)
  --enable-mingw          Build Berkeley DB for MinGW.
  --enable-o_direct       Enable the O_DIRECT flag for direct I/O.
```

```
  --enable-posixmutexes   Force use of POSIX standard mutexes.

  --enable-smallbuild     Build small footprint version of the library.
  --enable-stl           Build STL API.
  --enable-tcl           Build Tcl API.
  --enable-test          Configure to run the test suite.
  --enable-uimutexes      Force use of Unix International mutexes.
  --enable-umrw          Mask harmless uninitialized memory read/writes.
  --enable-shared[=PKGS]  build shared libraries [default=yes]
  --enable-static[=PKGS]  build static libraries [default=yes]
  --enable-fast-install[=PKGS]
                        optimize for fast installation [default=yes]
  --disable-libtool-lock  avoid locking (might break parallel builds)
  --disable-largefile     omit support for large files

Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --with-mutex=MUTEX      Select non-default mutex implementation.
  --with-mutexalign=ALIGNMENT
                        Obsolete; use DbEnv::mutex_set_align instead.
  --with-tcl=DIR          Directory location of tclConfig.sh.
  --with-uniquename=NAME  Build a uniquely named library.
  --with-pic             try to use only PIC/non-PIC objects [default=use
                        both]
  --with-gnu-ld          assume the C compiler uses GNU ld [default=no]
```

查看上面的信息，设置交叉编译的 configure 参数如下：

```
# ../dist/configure CC=arm-linux-gcc --host=arm-linux --enable-
mutexsupport \
--exec-prefix=/usr/local/BerkeleyDB ARM
--docdir=/usr/local/BerkeleyDB ARM/doc \
--includedir=/usr/local/BerkeleyDB ARM/include  --enable-cxx
# make
# make install
```

在目录/usr/local/BerkeleyDB_ARM 下生成 4 个文件夹：

```
bin  include  lib  doc
```

查看安装的 lib 库文件：

```
libdb-4.8.a   libdb-4.8.so  libdb.a          libdb_cxx-4.8.la
                                             libdb_cxx-4.so  libdb_cxx.so
libdb-4.8.la  libdb-4.so  libdb_cxx-4.8.a  libdb_cxx-4.8.so  libdb_cxx.a
                                             libdb.so
```

# 15.3　使用 Berkeley DB 数据库

前面已经介绍 Berkeley DB 数据库的基本概念和编译安装过程。本节就介绍如何使用安装好的 Berkeley DB 数据库。这里以 C++库为例介绍其使用方法。

## 15.3.1　代码分析

以目录 examples_cxx 下的文件 AccessExample.cpp 为例，说明如何编写程序操作

Berkeley DB 数据库。前面介绍数据库基本概念时，基本上介绍了例子中的数据库操作，读者不明白的地方可以参考 15.1 节或者参考 Berkeley DB 数据库的 C++ API 文档。

AccessExample 主要实现 Key/Data 存储和从数据库取数据的功能。程序的实现主要分为 3 部分：类定义、主函数和访问数据库。下面分别介绍这 3 部分。

### 1．类的定义部分

AccessExample 的定义部分主要定义了访问数据库的方法和构造函数。其代码如下：

```
#include <db cxx.h>
#define DATABASE    "access.db"
class AccessExample                                     //定义测试类
{
public:
    AccessExample();
    void run(bool removeExistingDatabase, const char *fileName);
                                                        //定义测试数据库方法

private:
    // no need for copy and assignment
    AccessExample(const AccessExample &);               //复制构造函数
    void operator = (const AccessExample &);            //赋值构造函数
};
```

### 2．主函数部分

主函数部分是程序的入口点，指定了运行时选择的数据库。调用 run()函数进行访问数据库。同时使用 try/catch 机制捕获访问数据库过程中的异常。

```
int main(int argc, char *argv[])
{
    int ch, rflag;
    const char *database;

    rflag = 0;
    while ((ch = getopt(argc, argv, "r")) != EOF)
        switch (ch) {
        case 'r':
            rflag = 1;
            break;
        case '?':
        default:
            return (usage());
        }
    argc -= optind;
    argv += optind;

    /*如果命令行后面跟上参数为空，则数据库为默认的数据库“access.db”，否则取参数作为
    数据库*/
    database = *argv == NULL ? DATABASE : argv[0];

    /*使用 try-catch 捕捉数据库操作过程中的异常*/
    try {
        AccessExample app;
        app.run((bool)(rflag == 1 ? true : false), database);
```

```
        return (EXIT_SUCCESS);
    }
    catch (DbException &dbe) {
        cerr << "AccessExample: " << dbe.what() << "\n";
        return (EXIT_FAILURE);
    }
}
```

### 3. 访问数据库部分

访问数据库部分主要完成构造数据库，打开数据库，向数据库中存储数据，以及遍历数据取数据，最后关闭数据库。

```
void AccessExample::run(bool removeExistingDatabase, const char *fileName)
{
    //移除旧的数据库
    if (removeExistingDatabase)
        (void)remove(fileName);

    //不需要环境句柄进行创建数据库对象
    Db db(0, 0);

    db.set_error_stream(&cerr);
    db.set_errpfx("AccessExample");
    db.set_pagesize(1024);       /* Page size: 1K. */
    db.set_cachesize(0, 32 * 1024, 0);
    //打开数据库 fileName，如果数据库 fileName 不存在则创建数据库
    db.open(NULL, fileName, NULL, DB_BTREE, DB_CREATE, 0664);

    /*插入键值对到数据库中，输入的字符串键，其逆序为值*/
    char buf[1024], rbuf[1024];
    char *p, *t;
    int ret;
    u_int32_t len;

    for (;;) {
        cout << "input> ";
        cout.flush();                      //刷新输出缓冲区到输出流

        cin.getline(buf, sizeof(buf));     //获取输入行
        if (cin.eof())                     //如果输入结束，Windwos 中 Ctrl+Z
                                              模拟文件结束，Linux 下 Ctrl+D 模
                                              拟文件结束
            break;

        if ((len = (u_int32_t)strlen(buf)) <= 0)
            continue;
        for (t = rbuf, p = buf + (len - 1); p >= buf;)         //逆序
            *t++ = *p--;
        *t++ = '\0';

        /*这里创建的两个对象 key 和 data 分别代表要存储的一个键值对的 key 和 data。把
        字节串的起始地址和长度传给了它们，Berkeley DB 即可得到这两个字节串*/
        Dbt key(buf, len + 1);
        Dbt data(rbuf, len + 1);

        /*保存 key 和 data，DB_NOOVERWRITE 表示：如果存在重复的 key 和 data，则不覆
```

```
        盖旧的 key 和 data*/
        ret = db.put(0, &key, &data, DB_NOOVERWRITE);
        if (ret == DB_KEYEXIST) {
            cout << "Key " << buf << " already exists.\n";
        }
    }
    cout << "\n";

    try {
        //建立游标用于遍历数据库
        Dbc *dbcp;
        db.cursor(NULL, &dbcp, 0);

        //遍历整个表打印 key/data 对
        Dbt key;
        Dbt data;
        while (dbcp->get(&key, &data, DB_NEXT) == 0) {
            char *key_string = (char *)key.get_data();
            char *data_string = (char *)data.get_data();
            cout << key_string << " : " << data_string << "\n";
        }
        /*关闭游标。因为游标稳定性导致游标所引用的页面被锁定，使用同一个数据库的其他
        进程或者线程无法访问这些页面*/
        dbcp->close();
    }
    /*捕获异常*/
    catch (DbException &dbe) {
        cerr << "AccessExample: " << dbe.what() << "\n";
    }
    /*关闭数据库*/
    db.close(0);
}
```

## 15.3.2　编译运行程序

AccessExample.cpp 使用了 C++ API。编译 AccessExample.cpp 的命令如下：

```
#  g++  -o  AccessExample  -I  /usr/local/BerkeleyDB.4.8/include/  -L
/usr/local/BerkeleyDB.4.8/lib/ AccessExample.cpp -lpthread -ldb_cxx
```

命令说明：

```
g++                                         //编译器
-o AccessExample                            //输出文件名
-I /usr/local/BerkeleyDB.4.8/include/       //指定头文件路径
-L /usr/local/BerkeleyDB.4.8/lib/           //指定库文件路径
AccessExample.cpp                           //源文件
-lpthread -ldbcxx               //指定库名字为 libpthread 和 libdb_cxx
```

编译完成后生成可执行文件 AccessExample。如果没有设置 lib 库的路径，运行 AccessExample 时会出现找不到库的错误。添加库的路径到文件/etc/ld.so.conf 后面，并且运行 ldconfig 使之生效。

```
# vi /etc/ld.so.conf
```

添加下面内容：

/usr/local/BerkeleyDB.4.8/lib

```
# ldconfig
```

运行 AccessExample，命令后面带参数，参数即为数据库名称，不带参数则数据库名称默认为 access.db。保存输入 key/data 对，并且打印整个表信息。运行执行结果如下：

```
# ./AccessExample
input> abc                              //输入 abc
input> bcd                              //输入 bcd
input> cde                              //输入 cde
input> efg                              //输入 efg
input> ghi                              //输入 ghi
input>                                  //按下 Ctrl+D 表示输入结束
abc : cba
bcd : dcb
cde : edc
efg : gfe
ghi : ihg
```

# 15.4　移植 Berkeley DB 数据库

本节中将会以实例讲解 Berkeley DB 设计类似电话本功能的查询、添加功能。在调试的中先在上位机中编译、调试后，再进行交叉编译，然后进行移植。

## 15.4.1　数据库设计

使用 Berkeley DB 数据库时，一个数据库中包含一个表。这个简单的表包含两个字段：用户姓名和号码。通过输入用户姓名和号码保存到数据库，然后遍历打印整个表信息。

数据库名称：user2num.db。

- Key：user。
- Data：num。

数据库的操作包括：

- 存数据操作 put()。
- 取数据操作 get()。

## 15.4.2　编写应用程序

根据上面的设计，编写代码时主要功能实现为存数据操作和取数据操作。本程序 PhoneBook.cpp 参考例子程序 AccessExample.cpp。下面是程序源代码。

```
//PhoneBook.cpp                                 //文件名
#include <sys/types.h>                          //添加的所有头文件
#include <iostream>
```

```
#include <iomanip>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <db_cxx.h>

//名字空间
using std::cin;
using std::cout;
using std::cerr;
using std::endl;

int  main()
{
    char *fileName;
    Db db(0, 0);                    //创建没有环境的数据库

    db.set_error_stream(&cerr);
    db.set_errpfx("PhoneBook");
    db.set_pagesize(1024);          //页大小为 1K
    db.set_cachesize(0, 32 * 1024, 0);

    fileName = "user2num.db";
    /*打开数据库 fileName。如果该数据库不存在则创建名字为 fileName 的数据库*/
    db.open(NULL, fileName, NULL, DB_BTREE, DB_CREATE, 0664);

    char user_buf[1024], num_buf[1024];
    char *p, *t;
    int ret;
    u_int32_t user_len, num_len;

    for (;;) {
            /*提示输入用户姓名*/
            cout << "input name> ";
            cout.flush();
            cin.getline(user_buf, sizeof(user_buf));
            /*提示输入对应用户的电话号码*/
            cout<< " input phone num>";
            cout.flush();
            cin.getline(num_buf, sizeof(num_buf));

            /*判断是否介绍输入，结束则跳出循环，否则继续输入*/
            if (cin.eof())
                    break;

            /*如果输入为空则不执行后续操作，即不存入数据库*/
            if ( ((user_len = (u_int32_t)strlen(user_buf)) <= 0) || ((num_len
             = (u_int32_t)strlen(num_buf)) <= 0) )
                    continue;

            /*创建的两个对象 key 和 data 分别代表要存储的一个键值对的 key 和 data。把
            字节串的起始地址和长度传给了它们，Berkeley DB 即可得到这两个字节串*/
            Dbt key(user_buf, user_len + 1);
            Dbt data(num_buf, num_len + 1);
```

```
            /*将输入的 Key 和 Data 对存入数据库，如果已经存在则不覆盖*/
            ret = db.put(0, &key, &data, DB_NOOVERWRITE);
            if (ret == DB_KEYEXIST) {
                    cout << "Key " << user_buf << " already exists.\n";
            }
    }
    cout << "\n";

    try {
       /*定义访问数据库的游标*/
       Dbc *dbcp;
       db.cursor(NULL, &dbcp, 0);

       /*遍历数据库打印姓名和号码*/
       Dbt key;
       Dbt data;
       while (dbcp->get(&key, &data, DB_NEXT) == 0) {
              char *key_string = (char *)key.get_data();
              char *data_string = (char *)data.get_data();
              cout << key_string << " : " << data_string << "\n";
       }
       /*关闭游标*/
       dbcp->close();
    }
    catch (DbException &dbe) {
            cerr << "AccessExample: " << dbe.what() << "\n";
    }

    db.close(0);
}
```

## 15.4.3　调试和交叉编译应用程序

在上位机中进行编译和调试，使用的是/usr/local/BerkeleyDB.4.8/目录下的头文件和库。编译的方法为：

```
# g++ -o PhoneBook -I /usr/local/BerkeleyDB.4.8/include/ -L /usr/local/
BerkeleyDB.4.8/lib/ PhoneBook.cpp -lpthread -ldb_cxx
# ./PhoneBook                                   //运行
input name> Tom                                 //输入姓名
 input phone num>13012345678                    //对应的电话号码
input name> Jack
 input phone num>13787654321
input name> Mary
 input phone num>13543216789
/*下面为打印的电话本信息*/
Jack : 13787654321
Mary : 13543216789
Tom : 13012345678
```

在上位机中调试通过后可以进行交叉编译和移植。交叉编译使用的是/usr/local/BerkeleyDB_ARM/目录下的头文件和库。编译方法为：

```
# arm-linux-g++ -o PhoneBook -I /usr/local/BerkeleyDB_ARM/include/ -L
/usr/local/BerkeleyDB_ARM/lib/ PhoneBook.cpp -lpthread -ldb_cxx
```

交叉编译后生成：PhoneBook。

### 15.4.4　数据库的移植和测试

移植时需要将交叉编译库/usr/local/BerkeleyDB_ARM/lib 目录的 libdb_cxx.so 和 libdb_cxx-4.8.so 两个文件复制到开发板/lib 目录下。将 PhoneBook 放在开发板/usr/bin 目录下，并且修改其权限。

```
# chmod 777 PhoneBook
#./PhoneBook
```

在开发板上运行和上位机中运行效果相同，在当前目录下也会生成 user2num.db 数据库，运行结果如图 15.1 所示。

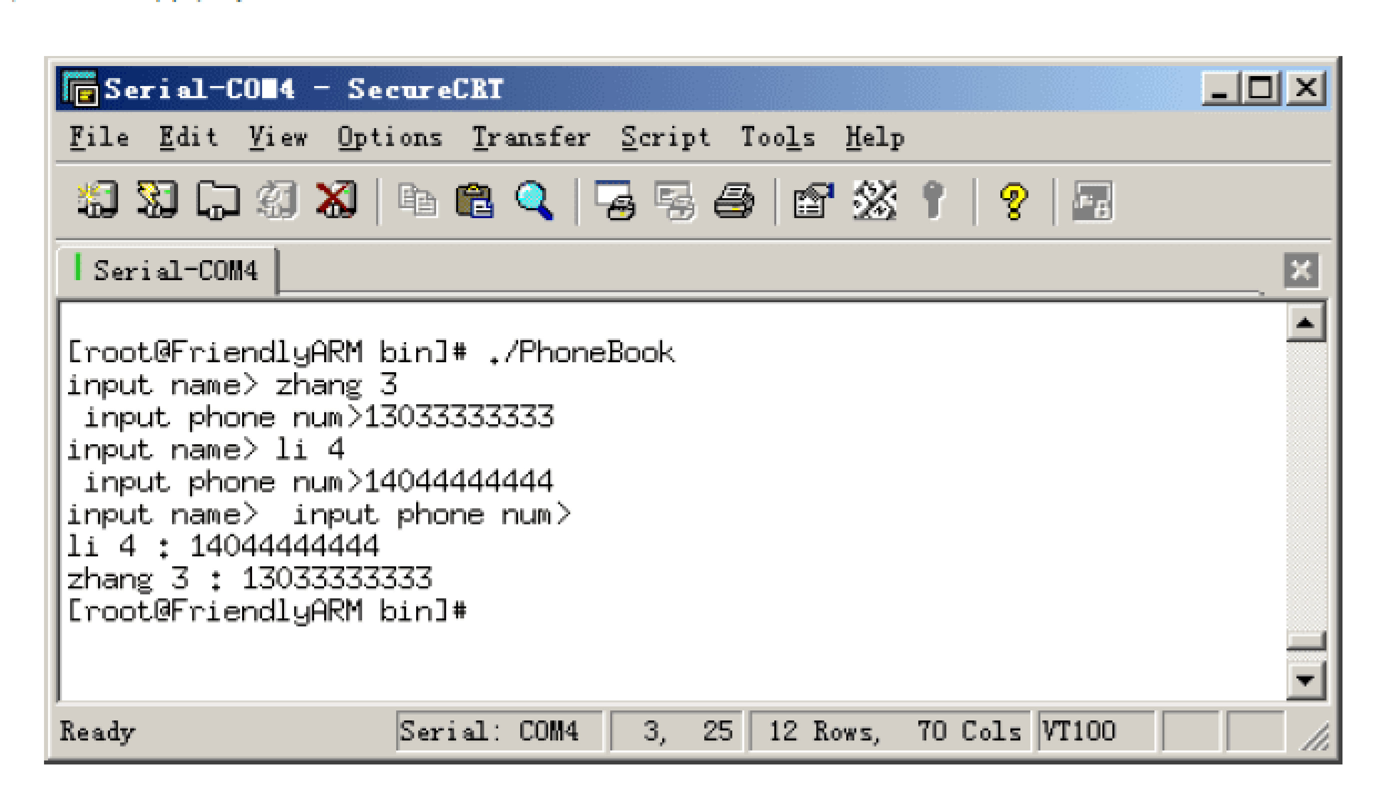



图 15.1　开发板上运行结果

## 15.5　小　　结

Berkeley DB 数据库在嵌入式系统中应用越来越多，本章的编译和移植过程都比较简单。在实际的项目中使用 Berkeley DB 数据库时，重点是在数据库的设计。掌握数据库的设计，将需求变为数据库的详细设计方案是读者将 Berkeley DB 数据库应用到嵌入式系统的基础。

# 第 16 章　嵌入式数据库 SQLite 移植

SQLite 是一个轻量级的数据库，非常适合用在嵌入式系统中。与 Berkeley DB 相比，它支持 SQL 语句，而且能在一个数据库中创建多个表。本章将主要介绍如何使用 SQLite 数据库及其接口，同时介绍其在上位机中的编译和安装，最后介绍如何将它移植到嵌入式系统中。

## 16.1　SQLite 支持的 SQL 语句

SQLite 支持大部分 SQL 语句，包括创建索引、创建表、创建视图、创建虚表、删除表、删除索引、删除视图、修改表等。本节将带读者回顾几个简单的 SQL 语句，为后面安装和使用 SQLite 做铺垫。如果读者熟悉 SQL 语句可以直接看 16.2 节。

### 16.1.1　数据定义语句

下面给出 SQL 标准中的数据定义语句种类，并且在括号中指明 SQLite 是否支持该语句。同时通过实例给出所支持语句的使用方法。

- ALTER DATABASE 语法（不支持）。
- ALTER TABLE 语法（支持），用于更改原有表的结构。
- CREATE DATABASE 语法（不支持）。
- CREATE INDEX 语法（支持），用于创建索引。
- CREATE TABLE 语法（支持），用于创建表。
- DROP DATABASE 语法（不支持）。
- DROP INDEX 语法（支持），用于删除索引。
- DROP TABLE 语法（支持），用于删除表。
- RENAME TABLE 语法（不支持）。

下面通过实例说明以上语法：

（1）创建表 student，该表包含 4 个列 id，name，sex 和 age。

```
create table student(id, name , sex, age );
```

（2）修改表 student，在表中增加 1 个列 address。

```
alter table student add address;
```

（3）创建索引 index_id，索引是根据表 student 的列 id 进行创建。

```
create index index_id on student (id);
```

（4）删除表中的索引 index_id，在 SQLite 数据库中索引不能重名，不同的表只能有一个名字为 index_id 的索引。删除的时候不需要指定是哪个表的索引。

```
drop index index_id;
```

### 16.1.2　数据操作语句

下面给出 SQL 标准中的数据操作语句种类，并且在括号中指明 SQLite 是否支持该语句。同时通过实例给出所支持语句的使用方法。

- ❑ DELETE 语法（支持）。
- ❑ DO 语法（不支持）。
- ❑ HANDLER 语法（不支持）。
- ❑ INSERT 语法（支持）。
- ❑ LOAD DATA INFILE 语法（不支持）。
- ❑ REPLACE 语法（支持）。
- ❑ SELECT 语法（支持）。
- ❑ Subquery 语法（不支持）。
- ❑ TRUNCATE 语法（不支持）。
- ❑ UPDATE 语法（支持）。

DELETE 语法，删除表中的一条或多条记录。INSERT 语法，向表中插入一条记录。

```
delete from student where id =1;
```

DELETE 命令用于从表中删除记录。命令包含 DELETE FROM 关键字及需要删除的记录所在的表名。若不使用 WHERE 子句，表中的所有行将全部被删除。否则仅删除符合条件的行。本句删除 id 为 1 的行。

```
insert into student values (11, 'DaSan', 'M', 20, 'beijing');
```

本句向表 student 中插入 id 字段为 11，name 字段为 DaSan，sex 字段为 M，age 字段为 20，address 字段为 beijing 的行。

```
select * from student;
```

select 语句从表 student 中查询所有项。

```
update student set age = 24 where age = 20;
```

update 语句更新所有字段 age 为 20 的值，将其更新为 age 为 24。

如果要对于 SQLite 的各种语法进行更深入的学习，可以参考 SQLite 的文档或者其他资料，本章重点不是 SQL 语句。下面将介绍如何在上位机中安装 SQLite，上面的语句在安装了 SQLite 后可以一一得到验证。

## 16.2　SQLite 数据库编译、安装和使用

SQLite 的安装过程也比较简单，但是笔者在安装最新版 sqlite-3.6.23 时出现了很多编

译错误。由于时间的关系没有深究这些错误的原因，而是最后选择 sqlite-3.6.17 安装成功。本机的环境 gcc 版本为 4.1.1。

## 16.2.1　安装 SQLite

安装 SQLite 的过程比较简单，下面列出安装 sqlite-3.6.17.tar.gz 的详细过程及安装过程的命令。

（1）复制 sqlite-3.6.17.tar.gz 到/usr/local 目录下。

（2）解压 sqlite-3.6.17.tar.gz。

```
#tar zxvf sqlite-3.6.17.tar.gz
```

（3）新建一个安装目录/usr/local/sqlite_x86。

```
#mkdir /usr/local/sqlite_x86
```

（4）进入解压目录/usr/local/sqlite-3.6.17 配置 SQLite，执行 configure 命令生成 Makefile 文件。

```
#cd /usr/local/sqlite-3.6.17
# ./configure -prefix=/usr/local/sqlite_x86
```

（5）执行 make 安装 SQLite。

```
# make
```

（6）执行 make install 将 SQLite 安装在/usr/local/sqlite_x86 路径下。

```
# make install
```

安装完成后进入/usr/local/sqlite_x86/目录查看安装文件。

```
# cd /usr/local/sqlite x86/
# ls
bin  include  lib                //安装目录下的文件
#cd bin
#ls
sqlite3                          //访问数据库的工具
```

（7）安装完成后，删除安装目录下的临时文件。

```
# rm -rf /usr/local/sqlite-3.6.17
```

## 16.2.2　利用 SQL 语句操作 SQLite 数据库

SQLite 安装完成后对 SQLite 进行测试，根据前面列出的 SQL 语句对 SQLite 支持的 SQL 语句进行测试。进入安装目录/usr/local/sqlite_x86/bin 测试工具 sqlite3。

```
#cd /usr/local/sqlite x86/bin
# ./sqlite3 testdb.db
```

创建数据库会出现下面的提示信息：

```
SQLite version 3.6.17
```

```
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

执行./sqlite3 testdb.db 后出现 SQLite 版本信息，并且提示用户输入 SQL 语句，下面为创建学生表及插入数据的 SQL 语句。

```
sqlite> create table student(id, name , sex, age );      //创建学生表
sqlite> insert into student values(1, 'Jack', 'M', 20); //向表中插入数据
sqlite> insert into student values(2, 'Tom', 'M', 21);
sqlite> insert into student values(3, 'Mary', 'W', 19);
sqlite> select * from student;                          //显示表的所有信息
```

显示的结果如下：

```
1|Jack|M|20
2|Tom|M|21
3|Mary|W|19
```

以下为更新表的操作，将学生表中 age=24 的项更新为 age=20。下面为输入的 SQL 语句。

```
sqlite> update student set age = 24 where age = 20;       //更新表
sqlite> select * from student;
```

上述操作的结果打印如下：

```
1|Jack|M|24
2|Tom|M|21
3|Mary|W|19
```

执行完对数据库的操作后，退出数据库使用.quit 命令。

```
.quit                                            //退出
```

## 16.2.3　利用 C 接口访问 SQLite 数据库

SQLite 为 C 语言提供的支持库位于安装目录下，通过提供的 C 接口可以对 SQLite 数据库进行操作。下面例程演示如何调用接口函数进行访问数据库。其主要过程与 16.2.2 节利用 SQL 语句访问数据库效果类似。给出了测试程序 test.c 的代码、编译过程和运行结果。

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

int main( void )
{
    sqlite3 *db=NULL;
    char *zErrMsg = 0;
    int rc;
    int i=0;

    rc = sqlite3_open("test.db", &db); //打开指定的数据库文件，如果不存在将创建
                                          一个同名的数据库文件
    if( rc )
```

```
    {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
    else
        printf("opened database test.db successfully!\n");


    //创建一个表，如果该表存在，则不创建，并给出提示信息，存储在 zErrMsg 中
    char *sql = "create table student(id, name , sex, age);" ;
    //执行 SQL 语句
    sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

    //插入数据
    sql = "insert into student values(1, 'Jack', 'M', 20);" ;
    //执行 SQL 语句
    sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
    //插入数据
    sql = "insert into student values(2, 'Tom', 'M', 21);" ;
    sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
    //插入数据
    sql = "insert into student values(3, 'Mary', 'W', 19);" ;
    sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
    //查询结果
    sql = "select * from student;" ;
    sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

    //查询数据
    /*
    int sqlite3_get_table(sqlite3*, const char *sql,char***result , int
    *nrow , int *ncolumn ,char **errmsg );
    result 中是以数组的形式存放所查询的数据，首先是表名，再是数据。
    nrow、ncolumn 分别为查询语句返回的结果集的行数、列数，没有查到结果时返回 0
    */
    int nrow = 0, ncolumn = 0;
    char **fristResult;                         //二维数组存放结果

    sql = "select * from student;" ;
    printf("\n");
    sqlite3_get_table( db , sql , &fristResult , &nrow , &ncolumn , &zErrMsg );

    //打印查询结果
    printf( "row:%d column=%d \n" , nrow , ncolumn );
    printf( "\nThe result of querying is : \n" );

    for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
    printf( "fristResult[%d] = %s\n", i , fristResult[i] );

    //释放 fristResult 的内存空间
    sqlite3_free_table( fristResult );

    sql = "update student set age = 24 where age = 20;" ;
    printf("\n");
    sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

    nrow = 0;
    ncolumn = 0;
```

```
    char **secondResult;                         //二维数组存放结果
    //查询数据
    sql = "select * from student;" ;
    sqlite3 get table( db , sql , &secondResult , &nrow , &ncolumn ,
&zErrMsg );

    //打印查询结果
    printf( "row:%d column=%d \n" , nrow , ncolumn );
    printf( "\nThe result of querying is : \n" );

    for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
    printf( "secondResult[%d] = %s\n", i , secondResult[i] );

    //释放 secondResult 的内存空间
    sqlite3 free table( secondResult );

    printf("\n");
    sqlite3_close(db);                           //关闭数据库
    return 0;
}
```

编译程序命令如下：

```
# gcc -o test -I /usr/local/sqlite x86/include -L /usr/local/sqlite x86/lib
test.c -lsqlite3 -static -lpthread
```

编译命令说明：

```
gcc                                    //指定编译器
-o test                                //输出文件名
-I /usr/local/ sqlite x86/include      //指定头文件路径
-L /usr/local/ sqlite_x86/lib          //指定库文件路径
test.c                                 //源文件
-lsqlite3 -static -lpthread            //指定库名字为 libpthread 和
                                         libsqlite3，且为静态方式编译
```

执行编译命令后会生成可执行文件 test，运行可执行文件。打印结果与执行 SQL 语句结果类似。

```
#./test
opened database test.db successfully!
row:3 column=4
The result of querying is :
fristResult[0] = id
fristResult[1] = name
fristResult[2] = sex
fristResult[3] = age
fristResult[4] = 1
fristResult[5] = Jack
fristResult[6] = M
fristResult[7] = 20
fristResult[8] = 2
fristResult[9] = Tom
fristResult[10] = M
fristResult[11] = 21
fristResult[12] = 3
fristResult[13] = Mary
fristResult[14] = W
fristResult[15] = 19
```

```
row:3 column=4

The result of querying is :
secondResult[0] = id
secondResult[1] = name
secondResult[2] = sex
secondResult[3] = age
secondResult[4] = 1
secondResult[5] = Jack
secondResult[6] = M
secondResult[7] = 24
secondResult[8] = 2
secondResult[9] = Tom
secondResult[10] = M
secondResult[11] = 21
secondResult[12] = 3
secondResult[13] = Mary
secondResult[14] = W
secondResult[15] = 19
```

**注意：**表的第一行被更新了，和 SQL 语句执行的效果相同。通过调用接口后生成了 test.db 数据库和 student 表，此时也可以通过 SQL 语句进行打印。通过 SQL 语句查看调用 API 生成表的结果留给读者自己试验。

# 16.3　移植 SQLite

移植 SQLite 主要包括交叉编译数据库工具和库文件，编译应用程序，移植编译好库文件和应用程序到开发板，运行结果。

## 16.3.1　交叉编译 SQLite

为了与 x86 安装相区别，首先建立安装文件的目录 sqlite_arm，将交叉编译好的库文件和工具安装在此目录下。

### 1．建立交叉编译、安装目录

在/usr/local 目录下建立 sqlite_arm 目录，命令如下：

```
#mkdir /usr/local/sqlite_arm
```

### 2．配置交叉编译参数

配置交叉编译、安装参数，包括设置安装目录，不安装 tcl 支持库，目标主机为 arm-linux。配置命令如下：

```
# cd /usr/local/sqlite-3.6.17
# ./configure --prefix=/usr/local/sqlite arm --disable-tcl --host=arm-
linux
```

### 3. 交叉编译和安装

执行完 configure 命令后会生成 Makefile 文件。执行 make 和 make install 进行编译和安装 Sqlite。命令如下：

```
#make
#make install
```

安装完成后同样会在 sqlite_arm 目录下生成 include、lib、bin 三个目录。在 bin 目录下有工具 sqlite3；在 lib 和 include 目录下对应生成库文件和头文件。

## 16.3.2　交叉编译应用程序

交叉编译应用程序是为了在开发板上验证 Sqlite 数据库在开发板上是否能正确运行。交叉编译的应用程序仍然是前面在上位机中使用的 test.c 程序。将该程序复制到/sqlite_arm/bin/目录下进行交叉编译。实际上不用专门放在该目录下，这里笔者是为了与上位机中的测试程序相区别。执行下面的命令进行交叉编译应用程序：

```
# arm-linux-gcc -o test -I /usr/local/sqlite arm/include -L /usr/local/
sqlite_arm/lib/ test.c -lsqlite3 -static -lpthread
```

命令说明：

```
arm-linux-gcc                              //指定交叉编译器
-o test                                    //输出文件名
-I /usr/local/sqlite_arm/include           //指定头文件路径
-L /usr/local/sqlite_arm/lib               //指定库文件路径
test.c                                     //源文件
-lsqlite3 -static -lpthread                //指定库名字为 libpthread 和
                                             libsqlite3，且为静态方式编译
```

如果应用程序没有错误就会在该目录下生成可执行文件 test，可以通过 file test 或者 readelf test –h 查看生成的文件的格式。

```
# file test
```

下面为生成文件的信息，指定运行平台为 ARM。

```
test: ELF 32-bit LSB executable, ARM, version 1 (SYSV), for GNU/Linux 2.6.14,
statically linked, for GNU/Linux 2.6.14, not stripped
```

# 16.4　移植 SQLite 数据库

在前面的一些章节中，一直使用的是动态库，本例编译和移植将使用静态库方式。在实际的开发中动态库方式比较常用，使用动态库使得应用程序比较小。在大型项目中，肯定存在很多不同的模块访问相同的库，采用动态库可以减少占用的空间。而在单个模块调试期间采用静态方式编译可减少移植时间。

### 16.4.1　文件移植

通过 FTP 工具将 16.3 节中/usr/local/sqlite_arm 目录下生成的 test 移植到开发板上，并且修改其执行权限。为了测试 SQL 语句是否可以在开发板上正常执行，还要移植库文件 libsqlite3.so.0 和 sqlite3。

```
#chmod +x test
#chmod +x sqlite3
```

🔔**注意：**如果只测试其 C/C++接口，只移植 test 文件就可以了。库文件 libsqlite3.so.0 放在/lib 目录下。

### 16.4.2　运行应用程序

运行应用程序测试 C/C++接口。使用命令./test 执行应用程序和上位机中运行结果一致，如图 16.1 所示。

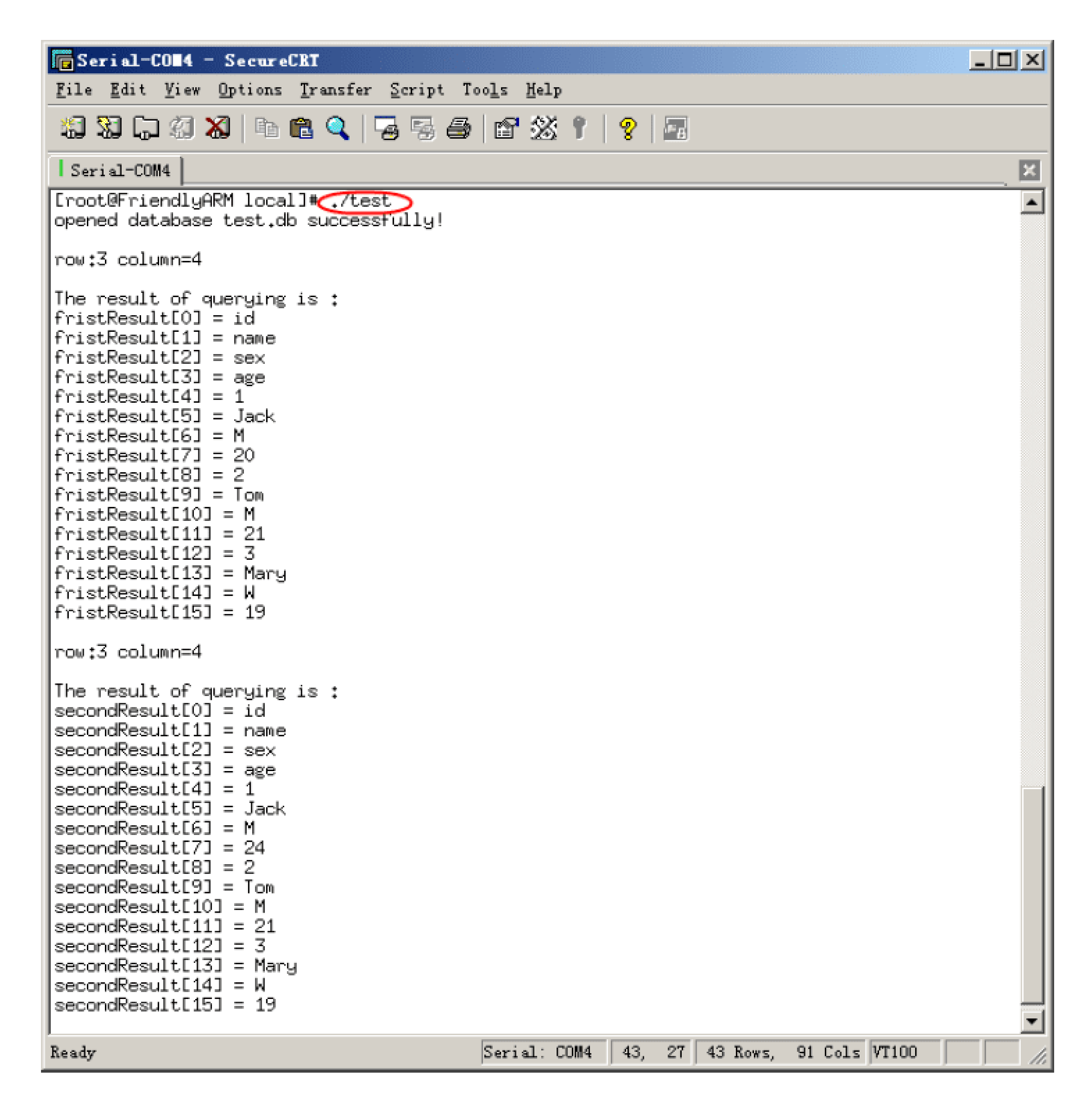



图 16.1　测试程序运行结果

### 16.4.3　测试 sqlite3

为了在开发板上运行 SQL 语句，应该测试工具 sqlite3 能否正确运行。使用命令./sqlite3

或者./sqlite3 test.db 进行测试，测试的具体命令如下，测试结果如图 16.2 所示。

```
# ./sqlite3 testdb.db            //创建数据库会出现下面的提示信息
```

执行./sqlite3 testdb.db 后出现 SQLite 版本信息，并且提示用户输入 SQL 语句。

SQLite version 3.6.17

Enter ".help" for instructions

Enter SQL statements terminated with a ";"

下面的 SQL 语句为创建教师表以及插入数据。执行语句如下：

```
sqlite> create table teacher(id, name , sex, age );      //创建教师表
sqlite> insert into teacher values(1, 'Jack', 'M', 40); //向表中插入数据
sqlite> insert into teacher values(2, 'Tom', 'M', 41);
sqlite> insert into teacher values(3, 'Mary', 'W', 49);
sqlite> select * from teacher;                           //显示表的所有信息
```

显示的结果如下：

1|Jack|M|40

2|Tom|M|41

3|Mary|W|49

下面的 SQL 语句作用是更新教师表，将 age=44 的项更新为 age=40。执行语句如下：

```
sqlite> update teacher set age = 44 where age = 40;  //更新表
sqlite> select * from teacher;
```

1|Jack|M|44

2|Tom|M|41

3|Mary|W|49

完成对数据的操作后，使用.quit 命令退出数据库。

```
.quit                                              //退出
```

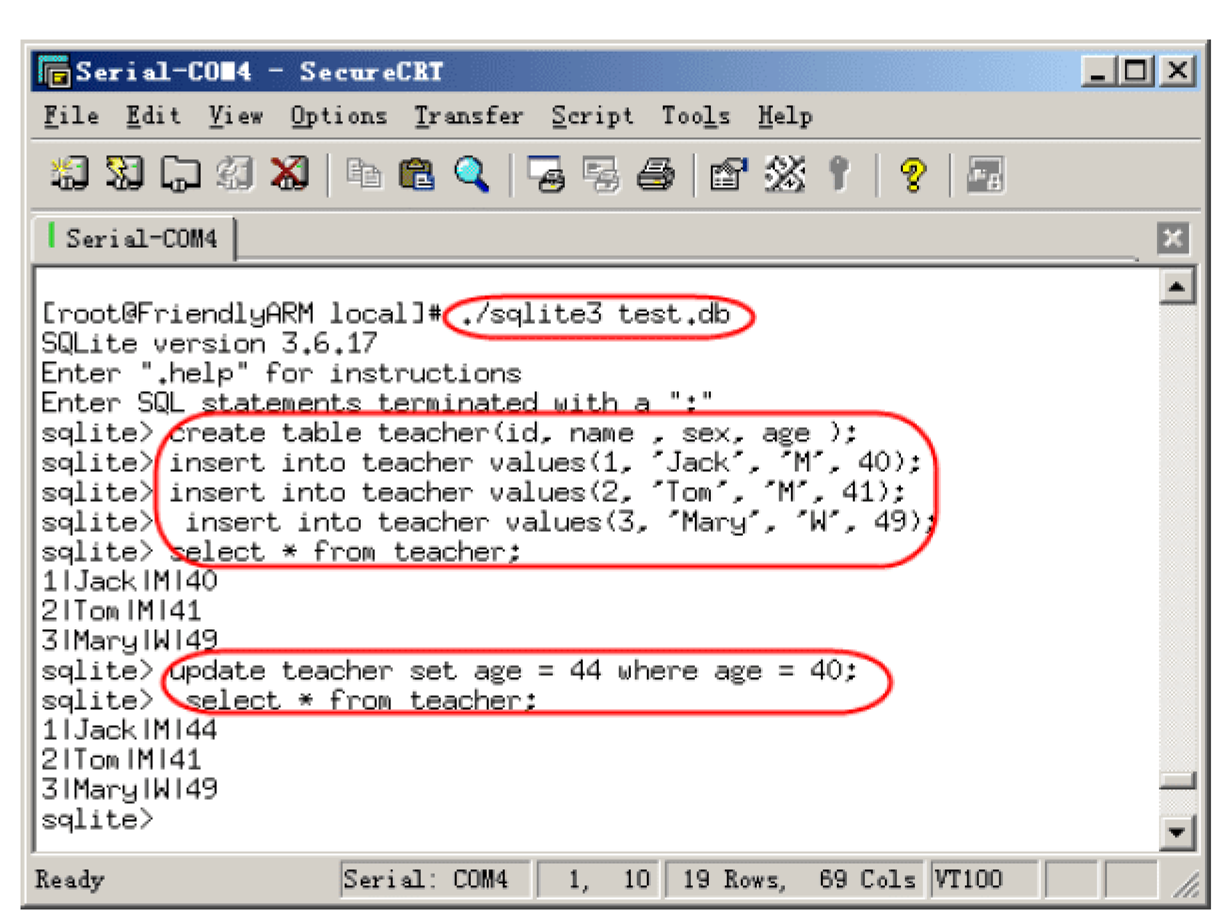



图 16.2　测试 SQL 语句

**注意：**红色圈住的部分为输入部分，与列出命令的黑体部分对应。对于其他的 SQL 语句读者可以自行进行测试。

## 16.5　小　　结

Sqlite 的移植与 Berkeley DB 移植类似，与 Berkeley DB 相比 Sqlite 多了对 SQL 语句的支持，可以通过 SQL 直接操作数据库。如果嵌入式系统中数据库不需要为多种语言如为 Java、C#等提供接口，可以采用 Sqlite 作为嵌入式数据库。使用 Sqlite 作为数据库可方便使用 SQL 对数据库进行维护。

# 第 17 章　嵌入式 Web 服务器 BOA 移植

早期的嵌入式设备维护的人机接口界面基本采用 C/S 模式。这种方式需要客户端安装特定的客户端程序，当维护界面升级后还要向客户端发布新的安装程序或者补丁。而采用 B/S 方式就不需要制作特定平台的客户端安装程序，也不需要因更新版本而向客户发布新的版本或补丁。BOA 是一款单任务的 Web 服务器，将 BOA 移植到嵌入式设备就能通过网络来维护设备，同时不需要关心操作系统和硬件平台，只需要终端设备安装浏览器。本章将主要介绍 BOA 的特点，编译过程，测试方法及移植过程。

## 17.1　BOA 介绍

BOA 是一款单任务的 HTTP 服务器，与其他 Web 服务器（IIS、APACHE、WEBLOGIC、WEBSPHERE、TOMCAT、JBOSS 等）相比，不同之处是当有连接请求到来时，它既不是为每个连接都单独创建进程，也不是采用复制自身进程处理多链接，而是通过建立 HTTP 请求列表来处理多路 HTTP 连接请求，同时它只为 CGI 程序创建新的进程，在最大程度上节省了系统资源，这对资源受限的嵌入式系统来说至关重要。同时它还具有自动生成目录、自动解压文件等功能，因此，BOA 具有很高的 HTTP 请求处理速度和效率，应用在嵌入式系统中具有很高的价值。

### 17.1.1　BOA 的功能

嵌入式 Web 服务器 BOA 完成的功能包括接收客户端请求、分析请求、响应请求、向客户端返回请求处理的结果等。BOA 的工作流程如下：

（1）修正 BOA 服务器的根目录。

（2）读配置文件（boa.conf）。

（3）写日志文件。

（4）初始化 Web 服务器，包括创建环境变量、创建 TCP 套接字、绑定端口、开始侦听、进入循环结构，以及等待和接收客户户的连接请求。

（5）当有客户端连接请求到达时，Web 服务器负责接收客户端请求，并保存相关请求信息。

（6）收到客户端的连接请求之后，Web 服务器分析客户端请求，解析出请求的方法、URL 目标、可选的查询信息及表单信息，同时根据客户端的请求做出相应的处理。

（7）Web 服务器处理完客户端的请求后，向客户端发送响应信息，最后关闭与客户机的 TCP 连接。

## 17.1.2　BOA 流程分析

查看 BOA 的流程可以通过查看 src/boa.c 文件中的 main()函数了解 BOA 的整个工作流程。下面将通过源码介绍 BOA 的主要工作流程。

### 1．修正BOA服务器的根目录

函数 fixup_server_root()判断 Web 服务器的根目录是否有效。如果 Web 服务器的根目录有效则指定根目录，否则打印错误信息并退出程序。

```
static void fixup_server_root()
{
    char *dirbuf;
    if (!server_root) {                 //如果没有指定根目录
#ifdef SERVER_ROOT                      //该宏在 defines.h 中被定义为"/etc/boa"
       //函数 strdup()功能为对参数目录字符串复制到新分配的字符指针，并将该指针作为返
         回值返回
       server_root = strdup(SERVER_ROOT);
        if (!server_root) {
            perror("strdup (SERVER_ROOT)");//分配空间和复制失败则打印信息并退出
            exit(1);
        }
#else                                   //如果没有在 defines.h 中定义为
                                         "/etc/boa"，则打印提示信息，并退出程序
        fputs("boa: don't know where server root is.  Please #define "
              "SERVER ROOT in boa.h\n"
              "and recompile, or use the -c command line option to "
              "specify it.\n", stderr);
        exit(1);
#endif
    }
    dirbuf = normalize_path(server_root);        //格式化路径
    free(server_root);                           //释放空间
    server_root = dirbuf;                        //成功指定根目录路径
}
```

### 2．读取配置文件

函数 read_config_files()用来读取配置文件信息，有关 Web 服务器的配置信息存放在文件 boa.conf 中。BOA 的配置信息包括 BOA 服务器监听的端口、绑定的 IP 地址、记载错误日志文件、设置存取日志文件等。

```
void read config files(void)
{
    char *temp;
    current_uid = getuid();
    yyin = fopen("boa.conf", "r");      //以只读方式打开配置信息文件 boa.conf

    if (!yyin) {                        //读取失败则打印打开文件失败信息
        fputs("Could not open boa.conf for reading.\n", stderr);
        exit(1);
    }
```

```
    if (!server_name) {               //如果没有指定服务器名字则指定服务器名字
        struct hostent *he;
        char temp_name[100];
        if (gethostname(temp_name, 100) == -1) {        //获得服务器名字
            perror("gethostname:");
            exit(1);
        }
        he = gethostbyname(temp_name);                  //获取主机
        if (he == NULL) {
            perror("gethostbyname:");
            exit(1);
        }
        server_name = strdup(he->h_name);               //获取主机名
        if (server_name == NULL) {
            perror("strdup:");
            exit(1);
        }
}
    tempdir = getenv("TMP");
    if (tempdir == NULL)
        tempdir = "/tmp";
//正确获得文档路径
    if (document_root) {
        temp = normalize_path(document_root);
        free(document_root);
        document_root = temp;
}
//获得错误日志路径
    if (error_log_name) {
        temp = normalize_path(error_log_name);
        free(error_log_name);
        error_log_name = temp;
}
//获得存取日志路径
    if (access_log_name) {
        temp = normalize_path(access_log_name);
        free(access_log_name);
        access_log_name = temp;
}
//获得公共网关接口日志路径
    if (cgi_log_name) {
        temp = normalize_path(cgi_log_name);
        free(cgi_log_name);
        cgi_log_name = temp;
    }
    if (dirmaker) {
        temp = normalize_path(dirmaker);
        free(dirmaker);
        dirmaker = temp;
    }
}
```

### 3. 写日志文件

函数 open_logs()打开日志文件并向文件中写日志。日志文件包括错误日志文件、存取日志文件、网关日志文件。

```
void open_logs(void)
```

```
{
    int error_log;
    if (error_log_name) {
        /*打开错误日志文件*/
        if (!(error_log = open_gen_fd(error_log_name))) {
            DIE("unable to open error log");
        }
        /*重定向错误输出到错误日志文件*/
        if (dup2(error_log, STDERR_FILENO) == -1) {
            DIE("unable to dup2 the error log");
        }
        close(error_log);
    }
    /*第 2 个参数为 F_SETFD 时，表示设置文件描述符标记。fcntl 文件锁有两种类型：建议
    性锁和强制性锁。系统默认 fcntl 都是建议性锁，当一个进程对文件加锁后，无论它是否释放
    所加的锁，只要文件关闭，内核都会自动释放加在文件上的建议性锁*/
    if (fcntl(STDERR_FILENO, F_SETFD, 1) == -1) {
        DIE("unable to fcntl the error log");
    }
    if (access_log_name) {
        /* 打开存取日志文件*/
        if (!(access_log = fopen_gen_fd(access_log_name, "w"))) {
            int errno_save = errno;
            fprintf(stderr, "Cannot open %s for logging: ",
                    access_log_name);
            errno = errno_save;
            perror("logfile open");
            exit(errno);
        }
        /*设置存取日志缓冲区*/
        setvbuf(access_log, (char *) NULL, _IOLBF, 0);
    } else
        access_log = NULL;

if (cgi_log_name) {
/*打开网关日志文件*/
        cgi_log_fd = open_gen_fd(cgi_log_name);
        if (cgi_log_fd == -1) {
            WARN("open cgi_log");
            free(cgi_log_name);                //打开网关日志文件失败，则释放资源
            cgi_log_name = NULL;
            cgi_log_fd = 0;
        } else {
           //打开成功则加锁
            if (fcntl(cgi_log_fd, F_SETFD, 1) == -1) {
                WARN("unable to set close-on-exec flag for cgi_log");
                //打开失败则关闭该文件，内核将自动释放加在该文件上的建议性锁
                close(cgi_log_fd);
                cgi_log_fd = 0;                //标识清零
                free(cgi_log_name);            //释放资源
                cgi_log_name = NULL;
            }
```

```
        }
    }
}
```

### 4．初始化Web服务器

函数 create_server_socket()是 Web 服务器的核心的函数。该函数的作用是建立服务端 TCP 套接字；然后将其转换为无阻塞套接字；并且给服务套接字加锁；函数 bind()用于建立套接字描述符与指定端口间的关联；并通过函数 listen()在该指定端口进行侦听，等待远程连接请求；当连接请求到达时，BOA 调用函数 get_request()获取请求信息，并通过调用函数 accept()为该请求建立一个连接；在建立连接之后，接收请求信息，同时对请求进行分析；当有 CGI 请求时，为 CGI 程序创建进程，并将结果通过管道发送输出。

```
static int create server socket(void)
{
    int server s;
    server_s = socket(SERVER_AF, SOCK_STREAM, IPPROTO_TCP);
                                             //创建 TCP 服务套接字
    if (server s == -1) {
        DIE("unable to create socket");
    }
    /*将服务套接字转换为无阻塞套接字*/
    if (set nonblock fd(server s) == -1) {
        DIE("fcntl: unable to set server socket to nonblocking");
    }
    /*加锁服务套接字*/
    if (fcntl(server s, F SETFD, 1) == -1) {
        DIE("can't set close-on-exec on server socket!");
    }
    /*当设置 TCP 套接口接收缓冲区的大小时，服务端应该在监听前进行设置*/
    if ((setsockopt(server_s, SOL_SOCKET, SO_REUSEADDR, (void *) &sock_opt,
                sizeof (sock opt))) == -1) {
        DIE("setsockopt");
    }
    /*绑定套接字*/
    if (bind server(server s, server ip) == -1) {
        DIE("unable to bind");
    }
    /*在指定端口进行监听，等待客户端的连接请求*/
    if (listen(server s, backlog) == -1) {
        DIE("unable to listen");
    }
    return server s;
}
```

## 17.1.3　BOA 配置信息

BOA 的配置信息都保存在文件 boa.conf 中，该目录默认是放在/etc/boa 目录下，BOA

默认在该路径下读取相关的所有配置信息。下面将介绍 boa.conf 文件中相关配置信息的内容：

- Port ：BOA 服务器监听的端口默认是 80。如果端口号小于 1024 时，则必须是 root 用户启动服务器。

```
Port 80
```

- Listen：指定绑定的 IP 地址。注释掉该参数时，将绑定所有的地址。

```
#Listen 192.68.0.5
```

- User：连接到服务器的客户端身份，可以是用户名或 UID。可以在文件/etc/passwd 中查看是否存在用户名 nobody。

```
User nobody
```

- Group：连接到服务器的客户端的组，可以是组名或 GID。可以在文件/etc/group 中查看是否存在组名为 nogroup 的组。

```
Group nogroup
```

- ErrorLog：该文件用于指定错误日志文件。如果路径没有以“/”开始，则指定其路径为相对于 ServerRoot 的路径。

```
ErrorLog /var/log/boa/error_log
```

- AccessLog：用于设置存取日志文件。

```
AccessLog /var/log/boa/access_log
```

- DocumentRoot：用于指定 HTML 文件的根目录。

```
DocumentRoot /var/www
```

- DirectoryIndex：指定预生成目录信息的文件，注释掉此变量表示将使用 DirectoryMaker 变量。这个变量也就是设置默认主页的文件名。访问 BOA 服务器主页时就是访问的 index.html 页面。

```
DirectoryIndex index.html
```

- KeepAliveMax：每个连接允许的请求数量。如果将此值设为 0，表示不限制请求的数目。这里表示允许请求的数量为 1000。

```
KeepAliveMax 1000
```

- KeepAliveTimeOut：在关闭持久连接前等待下一个请求的秒数。

```
KeepAliveTimeout 10
```

- CGIPat：CGI 程序的环境变量。

```
CGIPath /bin:/usr/bin:/usr/local/bin
```

- ScriptAlias：指定服务端脚本路径的虚拟路径。

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
```

在部署 Web 服务器时主要是对该配置文件进行配置，以及设置该配置中指定文件的路径和相关文件。在运行服务器的时候将给出 Web 服务器的具体配置。

# 17.2　BOA 编译和 HTML 页面测试

移植前在上位机中编译和测试 BOA。下面将详细介绍编译 BOA 的过程。本节介绍编译过程时，将会按照遇到错误解决编译错误的顺序进行介绍。

## 17.2.1　编译 BOA 源代码

BOA 的源代码文件最新稳定版本为 boa-0.94.13.tar.gz。BOA 服务器的源代码在解压后的 src 目录下。这里在编译的上位机是 FC6，不同版本的上位机可能会出现不同的编译错误，下面是其详细的编译过程，提供给读者在具体编译的时候作为参考。

将 boa-0.94.13.tar.gz 源码复制在/usr/local 目录下，进行解压。

```
# tar zxvf boa-0.94.13.tar.gz
```

进入 src 目录，使用 configure 命令生成 Makefile 文件。

```
# cd boa-0.94.13/src/
# ./configure
# make
```

执行编译时，遇到下面的编译错误：

```
util.c:100:1: 错误：毗连“t”和“->”不能给出一个有效的预处理标识符
make: *** [util.o] 错误 1
```

上面的错误是有关预处理的错误，在进入文件 util.c 找到第 100 行，可以发现与预处理有关的内容，宏 TIMEZONE_OFFSET。

```
time_offset = TIMEZONE_OFFSET(t);
```

跟踪 TIMEZONE_OFFSET 的定义，找到该宏在文件 compat.h 中定义。下面是该宏的定义

```
#ifdef HAVE_TM_GMTOFF
#define TIMEZONE_OFFSET(foo) foo##->tm_gmtoff
#else
#define TIMEZONE_OFFSET(foo) timezone
#endif
```

根据错误提示信息，修改上述宏定义为：

```
#ifdef HAVE_TM_GMTOFF
#define TIMEZONE_OFFSET(foo) foo->tm_gmtoff
#else
#define TIMEZONE_OFFSET(foo) timezone
#endif
```

修改该宏定义后，再执行 make 进行编译，在 src 目录下生成 boa 可执行程序。

### 17.2.2　设置 BOA 配置信息

BOA 配置信息存放在 boa.conf 中，BOA 读取配置信息的时候默认路径是从/etc/boa。因此运行 boa 时需要建立目录/etc/boa，并将配置信息放在该目录下。也可修改代码中的宏定义 SERVER_ROOT（在文件 defines.h 中）。

建立正确的配置文件路径，并复制配置文件到该目录下。

```
# mkdir /etc/boa
# cp boa.conf /etc/boa
```

建立日志目录/var/log/boa。

```
#mkdir /var/log/boa
```

对配置信息的修改如下：

（1）文件/etc/group 中不存在组名为 nogroup 的组。

```
Group nogroup
```

修改为：

```
Group 0
```

（2）CGI 程序的环境变量。

```
CGIPath /bin:/usr/bin:/usr/local/bin
```

修改为：

```
CGIPath /bin:/usr/bin:/var/www/cgi-bin
```

（3）DirectoryIndex：不指定预生成目录信息的文件。

```
#DirectoryMaker /usr/lib/boa/boa_indexer              //注释掉该句
```

（4）指定服务端脚本路径的虚拟路径到/var/www 目录下。

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
```

修改为：

```
ScriptAlias /cgi-bin/ /var/www/cgi-bin/
```

（5）取消对 Server Name 的注释。

```
Server Name www.your.org.here
```

**注意：**默认情况下是注释了 ServerName，这样在运行 boa 时，会出现“gethostbyname::　No such file or directory”或者“get has tbyname::Success”等异常，出现无法问现象。

### 17.2.3　测试 BOA

测试 BOA 主要分为 3 步：编写测试页面，启动 Web 服务器，执行测试。根据配置文件中的信息可知，测试页面放在/var/www 目录下，文件名为 index.html。

### 1．编写测试主页index.html

在测试的过程中发现页面在显示中文的时候有乱码，因此，为解决中文乱码问题为测试页面添加中文字符编码设置，放在 index.html 的开头。

```
<%@ page contentType="text/html; charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

下面是测试页面的主要代码。

```
<html>
    <title>
        boa test page!
    </title>
    <head>
        <font color="#cc2200"><b></b>欢迎大家测试 BOA 服务器</font><p>
    </head>
    <body>
        这里是 BOA 服务器测试主页(http+BOA 服务器 ip 地址)<p>
        测试方法在浏览器中输入，BOA 服务器的 IP 然后回车。<p>
        <font style="background-color: #808080">http://192.168.217.128<
        /font><p>
        如果直接在 Linux 上位机中进行测试可以直接在浏览器中输入下面地址并回车。<p>
        <font style="background-color: #808080">http://127.0.0.1<
        /font><p>
    </body>
</html>
```

### 2．启动Web服务器

执行./boa 运行 Web 服务器。在进行页面访问测试之前，首先可以通过 ps 命令查看进程中是否存在 boa 进程。

```
#./boa
#ps
```

如果 BOA Web 服务器没有正常起来，可以在/var/log/boa 目录中查看 error_log 文件。如果在运行 boa 时出现错误。该错误会在错误日志中详细记录错误的原因。

```
boa.c:226 - icky Linux kernel bug!: Success
```

解决上面的错误，可以在 boa.c 文件中注释掉包含该行信息的相关代码，修改后重新编译并运行 boa Web 服务器。

```
/*
if (setuid(0) != -1) {
          DIE("icky Linux kernel bug!");
       }
*/
```

### 3．启动Web服务器

如果通过 ps 命令可以查看到 boa 已经运行起来了，可以在 Linux 服务器端的浏览器中通过输入 http://127.0.0.1 或者在客户端的浏览器中输入 http://192.168.217.128（服务器的 IP 地址）。

在浏览器中能正确显示测试主页，如果在服务器端对页面进行了修改，然后保存后，客户端只要刷新就能获得更新后的信息。

## 17.3　CGI 脚本测试

HTML 页面测试通过后，CGI 脚本的测试相对就容易很多。CGI 脚本的测试也包括三个部分：编写测试代码，编译测试代码，执行测试。

### 17.3.1　编写测试代码

CGI 的文件应该放在目录/var/www/cgi-bin 下，在该目录下编写 hello.c 文件，该测试文件内容为打印“Hello,World.”。测试文件的代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
        printf("Content-type: text/html\n\n");
        printf("<html>\n");
        printf("<head><title>CGI Output</title></head>\n");
        printf("<body>\n");
        printf("<h1>Hello,world.</h1>\n");
        printf("<body>\n");
        printf("</html>\n");
        exit(0);
}
```

### 17.3.2　编译测试程序

将 hello.c 文件编译生成 hello.cgi 文件。编译命令如下：

```
# gcc -o hello.cgi hello.c
```

编译生成 hello.cgi 后，如果该文件不是在目录/var/www/cgi-bin 下，则将 hello.cgi 文件复制到该目录下。

### 17.3.3　测试 CGI 脚本

打开客户端的浏览器，在浏览器中输入下面的地址进行访问。

```
http://192.168.217.128/cgi-bin/hello.cgi
```

在浏览器中正确显示“Hello,World.”。

## 17.4　BOA 交叉编译与移植

本节将介绍如何在嵌入式产品中应用 BOA，在嵌入式产品中使用 BOA 需要对其进行交叉编译、配置、编写 HTML 页面、编写 CGI、部署上述文件到相应的目录。

### 17.4.1　交叉编译 BOA

进入/usr/local/boa-0.94.13/src 目录对 BOA 进行交叉编译，这里使用的交叉编译器为 arm-linux-gcc-4.3.2。编译的过程如下：

（1）在上位机中调试的时候已经通过 configure 命令生成了 Makefile 文件，这里只需要对生成的 Makefile 文件进行修改。在 Makefile 文件中将 gcc 改为 arm-linux-gcc，将 gcc –E 改为 arm-linux-gcc –E。修改后保存然后再进行编译。

```
#cd /usr/local/boa-0.94.13/src
#vi Makefile
gcc 改为 arm-linux-gcc
gcc -E 改为 arm-linux-gcc -E
# make
```

**注意：**如果在执行 make 时，出现：make: Nothing to be done for `all'.，则表示已经存在 boa，也就是前面生成的 X86 平台的 BOA。可以通过 make clean 命令进行清除，然后再执行 make 进行编译。

（2）编译完成后，通过 file 命令对生成的执行文件进行查看。确认生成的是 ARM 平台格式的文件。

```
#file boa
```

查看该文件的属性如下：

```
boa: ELF 32-bit LSB executable, ARM, version 1 (SYSV), for GNU/Linux 2.6.14,
dynamically linked (uses shared libs), for GNU/Linux 2.6.14, not stripped
```

该信息内容非常丰富，表示生成的 BOA 为可执行文件，运行的平台为 ARM 体系结构，使用的是动态链接库，含有调试信息，而且还包括大小端和 GUN 版本信息。此时生成的 BOA 文件大小为 200K 左右，如果去除调试信息，BOA 文件的大小为 60K 左右。通过下面命令去除调试信息：

```
# arm-linux-strip boa
```

### 17.4.2　准备测试程序

测试程序包括测试的 HTML 页面和 CGI 程序，HTML 和 CGI 程序，使用在上位机中测试的程序。这里只需要对 CGI 程序进行交叉编译即可以使用在 ARM 平台上。

```
# arm-linux-gcc -o hello.cgi hello.c
```

同样，执行完成后对生成 hello.cgi 查看其文件信息。

### 17.4.3　配置 BOA

配置 BOA 时，主要包括创建相关的目录和将文件放在相应的目录中。在向 mini2440

上移植 BOA 的过程中，发现文件系统中已经有了 BOA，并且自动启动。配置文件 boa.conf 的路径与本例不变，放在/etc/boa 目录下。使用 ps 查看进程，找到 BOA 所在的目录/usr/sbin，将该文件删除，使用本例编译的程序。将 BOA 程序移植到开发板上时注意修改其权限为可执行。

（1）对原配置文件 boa.conf 基本不作修改，其内容如下：

```
Port 80
User root
Group root
ErrorLog /dev/console
AccessLog /dev/null
ServerName friendly-arm
DocumentRoot /www
DirectoryIndex index.html
KeepAliveMax 1000
KeepAliveTimeout 10
MimeTypes /etc/mime.types
DefaultType text/plain
CGIPath /www
AddType application/x-httpd-cgi cgi
```

（2）在原有目录/www 下没有 index.html 文件，则将该目录下存在的文件 leds.html 改名为 index.html。因为默认的主页名字为 index.html。

```
#cd /www
#mv leds.html index.html
```

（3）将主机中的文件/etc/mime.types 复制到开发板中对应的/etc 目录下。

### 17.4.4　测试

将编译好的程序替代旧的 BOA 程序后，重新启动 mini2440，在控制台打印 BOA 启动消息：

```
boa: server version Boa/0.94.13
boa: server built Apr  5 2010 at 21:52:19.
boa: starting server pid=459, port 80
```

（1）测试 HTML，在上位机浏览器中输入 http://192.168.1.230（为开发板的 IP 地址，与上位机在同一个 IP 段，上位机的 IP 为 192.168.1.199）。

在上位机中可以正确显示网页。在开发板的终端显示：

```
request from 192.168.1.199 "GET /favicon.ico HTTP/1.1" ("/www/favicon.ico"):
document open: No such file or directory
```

上面表明服务器已经正确收到请求 request 的消息，并且能正确解析。

（2）测试 CGI，在上位机浏览器中输入 http://192.168.1.230/ leds.cgi。浏览器中正确显示 leds.cgi 的页面。

## 17.5　BOA 与 SQLite 结合

在前面章节中，已经介绍了两种数据库 Berkeley DB 和 SQLite 的移植和使用。其维护

过程是通过终端来维护，这种方式在实际产品中主要用于前期的开发，在产品的运行阶段由于条件的限制对嵌入式系统应该采用远程维护的方式。本节将通过实例介绍通过 BOA Web 服务 CGI 接口维护和管理嵌入式产品中的数据库。

### 17.5.1　通过 CGI 程序访问 SQLite

SQLite 提供了 C 语言访问的接口，通过采用 C 语言程序访问数据库，将该访问数据库的操作编译成 CGI 程序部署在 BOA 的 CGI 路径下，远程维护人员通过调用此 CGI 程序就能实现远程维护数据库的目的。

下面是例子程序，其代码主要分为两部分：一部分是通过调用 C 接口对数据库进行创建、修改等维护工作；另一部分是通过 HTML 页面将结果返回给远程访问者。具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

int main( void )
{
    sqlite3 *db=NULL;
    char *zErrMsg = 0;
    int rc;
    int i=0;

    rc = sqlite3 open("test.db", &db);
                //打开指定的数据库文件,如果不存在将创建一个同名的数据库文件
    if( rc )
    {
        fprintf(stderr, "Can't open database: %s\n", sqlite3 errmsg(db));
        sqlite3 close(db);
        exit(1);
    }
    else
        printf("opened database test.db successfully!\n");

    //创建一个表，如果该表存在，则不创建，并给出提示信息，存储在 zErrMsg 中
    char *sql = "create table student(id, name , sex, age);" ;
    //执行 SQL 语句
    sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

    //插入数据
    sql = "insert into student values(1, 'Jack', 'M', 20);" ;
    //执行 SQL 语句
    sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
    //插入数据
    sql = "insert into student values(2, 'Tom', 'M', 21);" ;
    sqlite3 exec( db , sql, 0 , 0 , &zErrMsg );
    //插入数据
    sql = "insert into student values(3, 'Mary', 'W', 19);" ;
    sqlite3 exec( db , sql, 0 , 0 , &zErrMsg );
    //查询结果
    sql = "select * from student;" ;
    sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
```

```
//查询数据
/*
int sqlite3_get_table(sqlite3*, const char *sql,char***result , int
*nrow , int *ncolumn ,char **errmsg );
result 中是以数组的形式存放所查询的数据，首先是表名，再是数据。
nrow ,ncolumn 分别为查询语句返回的结果集的行数、列数，没有查到结果时返回 0
*/
int nrow = 0, ncolumn = 0;
char **fristResult;//二维数组存放结果

sql = "select * from student;" ;
printf("\n");
sqlite3 get table( db , sql , &fristResult , &nrow , &ncolumn , &zErrMsg );

//打印查询结果
printf( "row:%d column=%d \n" , nrow , ncolumn );
printf( "\nThe result of querying is : \n" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
printf( "fristResult[%d] = %s\n", i , fristResult[i] );

//释放掉 fristResult 的内存空间
sqlite3_free_table( fristResult );

sql = "update student set age = 24 where age = 20;" ;
printf("\n");
sqlite3 exec( db , sql, 0 , 0 , &zErrMsg );

nrow = 0;
ncolumn = 0;
char **secondResult;                //二维数组存放结果
//查询数据
sql = "select * from student;" ;
sqlite3 get table( db , sql , &secondResult , &nrow , &ncolumn ,
&zErrMsg );
//通过 CGI 将结果返回给远程操作者
 printf("<%@ page contentType=\"text/html; charset=gb2312\"%>");
                                    //设置编码方案，解决中文乱码
 printf("<html>\n");
 printf("<head><title>CGI Output</title></head>\n");
 printf("<body>\n");
 printf("<h1>Access SQLite Database by CGI of Boa</h1>\n");
 printf("<p>\n");
 printf("<p>\n");
 //打印查询结果
 printf( "row:%d column=%d \n" , nrow , ncolumn );
 printf( "\nThe result of querying is : \n" );

 for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ )
    printf( "secondResult[%d] = %s\n", i , secondResult[i] );

 printf("<body>\n");
 printf("</html>\n");
//释放掉 secondResult 的内存空间
sqlite3 free table( secondResult );

printf("\n");
```

```
    sqlite3_close(db);                  //关闭数据库
    return 0;
}
```

## 17.5.2　编译和测试

编译 SQLite 程序时，需要 SQLite 接口的头文件和库文件支持。对于 SQLite 的编译和安装过程，本节不再介绍，如果还没有安装 SQLite 的读者，可以参照 SQLite 数据库的移植对 SQLite 进行编译和安装。

### 1．CGI程序的编译和部署

将上述代码起名为 sqlite.c 放在 BOA 服务器的 CGI 路径下，对于本机的路径为/var/www/cgi-bin。然后使用下面的命令对其进行编译：

```
#gcc -o sqlite.cgi -I /usr/local/sqlite x86/include -L /usr/local/sqlite
x86/lib sqlite.c -lsqlite3 -static -lpthread
```

/usr/local/sqlite_x86 是本机 SQLite 的安装目录。编译完成后在/var/www/cgi-bin 目录下生成了 sqlite.cgi 文件，同时也完成了部署。

### 2．测试sqlite.cgi

在 Windows 的浏览器中输入 http://192.168.217.128/cgi-bin/sqlite.cgi，来访问虚拟机（虚拟机的 IP 地址为 192.168.217.128）下的 CGI 程序。测试结果如图 17.1 所示。

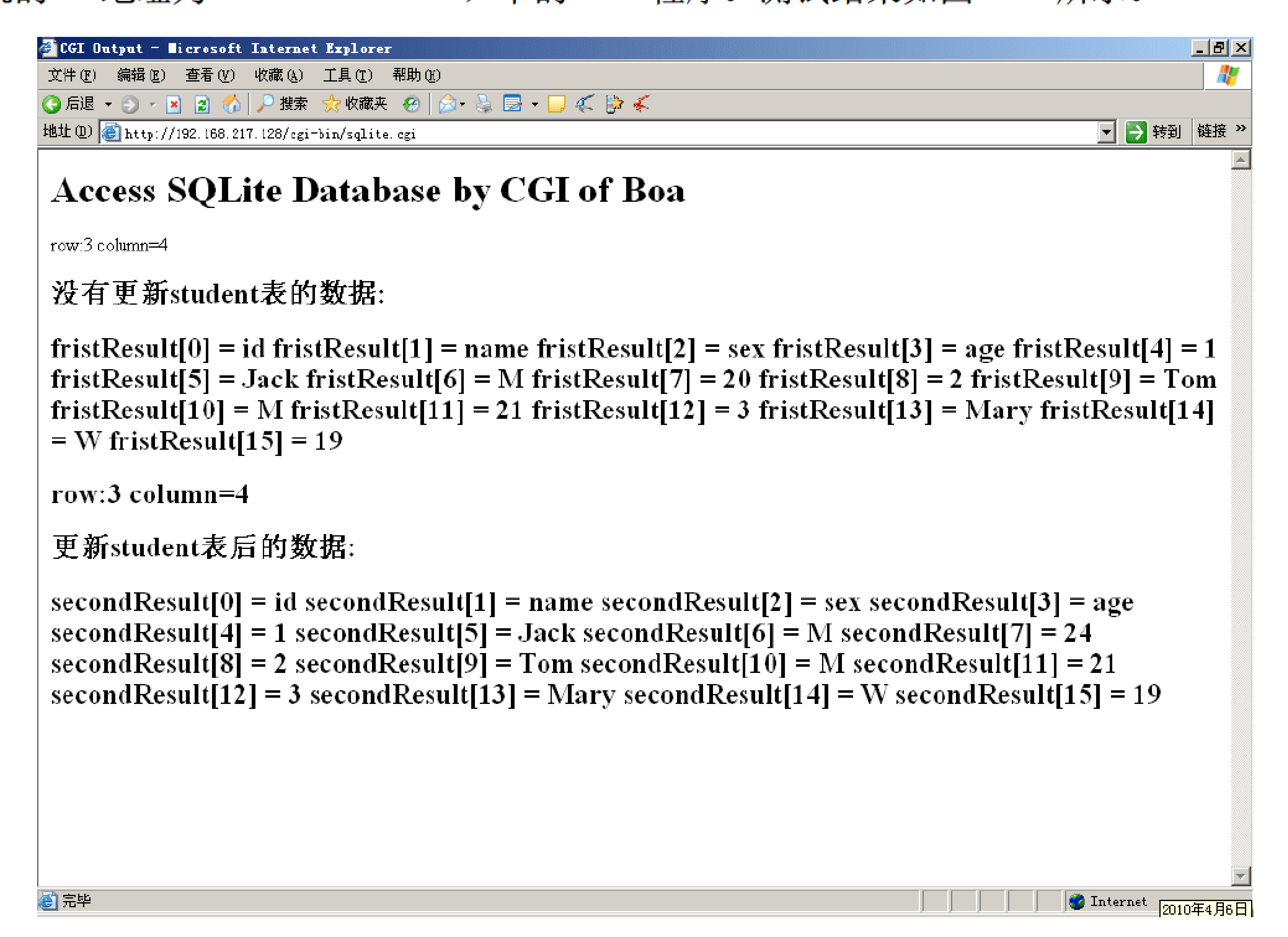



图 17.1　通过 CGI 访问 SQLite 数据库

**注意：**在嵌入式产品中结合使用 SQLite 和 BOA 具有很高的效率，实现起来也比较容易。对于其移植过程，请读者结合前面的内容自己动手体会。

## 17.6　小　　结

BOA 在嵌入式方面的应用非常简单有效，其编译和移植过程也比较简单。读者熟练掌握 BOA 的源代码和 SQLite 的源代码后，可以在自己的项目中进行灵活运用。另外，BOA 也可以和 Berkeley DB 数据库结合。本章的重点和难点是 BOA 的流程分析，比较实用并且很多市场前景是与嵌入式数据库的结合使用，读者可以编译更好的 CGI 程序维护远程的嵌入式数据库。

# 第 18 章　嵌入式 Web 服务器 Thttpd 移植

Thttpd 是一个简单的、小型的、可移植的、快速及安全的 HTTP 服务器。正因为它具有这些特点，将其应用在资源受限的嵌入式产品中非常合适。本章将介绍其编译、调试、使用和移植过程。

## 18.1　Thttpd 介绍

Thttpd 是一款小而快，且安全的 HTTP 服务器。下面将通过分析源代码介绍 Thttpd 工作过程。本章介绍的 thttpd 是基于 thttpd-2.25b 版本进行的。

### 18.1.1　Web 服务器比较

一般有 3 种常用 Web 服务器：Httpd、Thttpd 和 Boa。Httpd 是最简单的一个 Web 服务器，它的功能最弱，不支持认证，不支持 CGI（Common Gateway Interface，通用网关接口）。Thttpd 和 Boa 都支持认证、CGI 等，功能都比较全。Boa 源代码开放、性能可靠、稳定性好，但是仅能作为一个单任务的 Web 服务器。所以，使用简单、小巧、易移植、快速和安全的 Thttpd 嵌入式 Web 服务器是一个明智的选择。

另外，还有几款嵌入式 Web 服务器：Lighttpd、Shttpd、Mathopd、Minihttpd、Appweb、Goahead。读者有兴趣可以自己动手编译后进行测试，然后应用在自己的项目中。

### 18.1.2　Thttpd 的特点

Thttpd 的特点是高效、安全，并且支持 URL 流量控制。基于这些特点其在嵌入式方面的应用很有前景。下面分别介绍其特点：

#### 1．安全性

安全性问题中最大的危险源并不是来自授权协议本身，而是取决于在使用授权协议时所指定的策略和程序。所以 Thttpd 在默认的状况下，仅运行于普通用户模式下，从而能够有效地禁止非授权的系统资源和数据的访问，同时 Thttpd 全面支持 HTTP 基本验证（RFC2617 HTTP Authentication），可有效解决安全性的问题。这一点正像很多人在使用 Windows 时都是使用管理员的身份登录，因此系统经常容易受到病毒和木马程序的袭击，如果将管理员权限改为普通用户访问权限，那么系统将安全得多。

### 2．高效性

Thttpd 对于并发请求不使用 fork()来创建子进程处理，而是采用多路复用（Multiplex）技术来实现。而通过 fork()的方式创建子进程是父进程的一个复制，两者是独立的，使用该方式时，当并发的请求增多时，系统的性能被迅速降低。Thttpd 采用多路复用技术，当并发请求增多时节省了资源，提高了系统效率。

### 3．流量控制

Thttpd 支持基于 URL 的文件流量限制，便于处理连续的视频流量。与 Apache 比较，随着请求频率增加、请求数量增加时，Thttpd 的优势变得更加明显。

## 18.1.3　Thttpd 核心代码分析

Thttpd 工作流程的主逻辑在 thttpd.c 文件的 main()函数中。该函数中描述了 Thttpd 服务建立服务、接收请求、处理请求、日志文件及断开连接的过程。下面分析建立 Web 服务器的核心函数 httpd_initialize()。

函数 httpd_initialize()在文件 libhttpd.c 中定义，用于初始化 Web 服务器，如成功则返回 httpd_server 类型指针指向建立的 Web 服务器。该函数主要为 Web 服务器分配资源，初始化监听套接字，初始化媒体类型表准备接收客户端的请求。该函数的具体定义在 libhttpd.c 中，其定义如下：

```
httpd_server*
httpd_initialize(
    char* hostname, httpd_sockaddr* sa4P, httpd_sockaddr* sa6P,
    unsigned short port, char* cgi_pattern, int cgi_limit, char* charset,
    char* p3p, int max_age, char* cwd, int no_log, FILE* logfp,
    int no_symlink_check, int vhost, int global_passwd, char* url_pattern,
    char* local_pattern, int no_empty_referers )
    {
    httpd_server* hs;
    static char ghnbuf[256];
    char* cp;

    check_options();

    hs = NEW( httpd_server, 1 );    //为准备建立的 Web 服务器分配资源，1 表示所建
                                     立 Web 服务器的个数
    if ( hs == (httpd_server*) 0 )  //分配失败退出，并在系统日志中记录
     {
     syslog( LOG_CRIT, "out of memory allocating an httpd_server" );
     return (httpd_server*) 0;
     }

    if ( hostname != (char*) 0 )
     {
     hs->binding_hostname = strdup( hostname );
                                    //设置 Web 服务器的 binding_hostname 字段
     if ( hs->binding_hostname == (char*) 0 )
                                    //设置失败退出，并在日志中记录
```

```
        {
        syslog( LOG_CRIT, "out of memory copying hostname" );
        return (httpd_server*) 0;
        }
    hs->server_hostname = hs->binding_hostname;
                                    //设置 Web 服务器的 server_hostname 字段
    }
    else
    {
    hs->binding_hostname = (char*) 0;
    hs->server_hostname = (char*) 0;
    if ( gethostname( ghnbuf, sizeof(ghnbuf) ) < 0 )
                                    //当主机名为空时，获取主机名
        ghnbuf[0] = '\0';
#ifdef SERVER_NAME_LIST
    if ( ghnbuf[0] != '\0' )
        hs->server_hostname = hostname_map( ghnbuf );
                                    //设置 Web 服务器的 server_hostname 字段
#endif /* SERVER_NAME_LIST */
    if ( hs->server_hostname == (char*) 0 )
        {
#ifdef SERVER_NAME
        //如果没有定义宏 SERVER_NAME_LIST 中，则采用宏 SERVER_NAME 的定义
        hs->server_hostname = SERVER_NAME;
#else /* SERVER_NAME */
        if ( ghnbuf[0] != '\0' )
         hs->server_hostname = ghnbuf;
#endif /* SERVER_NAME */
        }
    }

    hs->port = port;                        //设置 Web 服务器的端口号
    if ( cgi_pattern == (char*) 0 )         //设置 Web 服务器的 cgi_pattern 字段
    hs->cgi_pattern = (char*) 0;
    else
    {
    /* Nuke any leading slashes. */
    if ( cgi_pattern[0] == '/' )
        ++cgi_pattern;
    hs->cgi_pattern = strdup( cgi_pattern );
    if ( hs->cgi_pattern == (char*) 0 )
                            //如果该字段设置失败，则退出程序并记录在系统日志文件中
        {
        syslog( LOG_CRIT, "out of memory copying cgi_pattern" );
        return (httpd_server*) 0;
        }
    /* Nuke any leading slashes in the cgi pattern. */
    while ( ( cp = strstr( hs->cgi_pattern, "|/" ) ) != (char*) 0 )
                            //返回“|/”在字段 cgi_pattern 中的位置
        (void) strcpy( cp + 1, cp + 2 );
                            //去掉 cgi_pattern 中的“|/”
    }
    hs->cgi_limit = cgi_limit;
                            //设置 Web 服务器中的字段 cgi_limit
    hs->cgi_count = 0;      //初始化 Web 服务器中的字段 cgi_count 为 0
    hs->charset = strdup( charset );
                            //设置 Web 服务器中的字段 charset
    hs->p3p = strdup( p3p );
```

```
                          //设置 Web 服务器中的字段 p3p
  hs->max_age = max_age;
                          //设置 Web 服务器中的字段 max_age
  hs->cwd = strdup( cwd );
                          //设置 Web 服务器中的字段 cwd
  if ( hs->cwd == (char*) 0 )
                          //字段 cwd 设置失败则退出并在系统日志中记录
   {
   syslog( LOG_CRIT, "out of memory copying cwd" );
   return (httpd_server*) 0;
   }
  if ( url_pattern == (char*) 0 )
                          //设置 Web 服务器中的字段 url_pattern
   hs->url_pattern = (char*) 0;
  else
   {
   hs->url_pattern = strdup( url_pattern );
   if ( hs->url_pattern == (char*) 0 )
       {
       syslog( LOG_CRIT, "out of memory copying url_pattern" );
       return (httpd_server*) 0;
       }
   }
  if ( local_pattern == (char*) 0 )
                          //设置 Web 服务器中的字段 local_pattern
   hs->local_pattern = (char*) 0;
  else
   {
   hs->local_pattern = strdup( local_pattern );
   if ( hs->local_pattern == (char*) 0 )
       {
       syslog( LOG_CRIT, "out of memory copying local_pattern" );
       return (httpd_server*) 0;
       }
   }
/*下面是对 Web 服务器的 no_log 、logfp 、logfp、vhost 、global_passwd、
no_empty_referers 字段的设置*/
  hs->no_log = no_log;
  hs->logfp = (FILE*) 0;
  httpd_set_logfp( hs, logfp );
  hs->c = no_symlink_check;
  hs->vhost = vhost;
  hs->global_passwd = global_passwd;
  hs->no_empty_referers = no_empty_referers;

  /*下面是对监听套接字初始化，如果优先考虑 IPv6*/
  if ( sa6P == (httpd_sockaddr*) 0 )
   hs->listen6_fd = -1;
  else
   hs->listen6_fd = initialize_listen_socket( sa6P );
  if ( sa4P == (httpd_sockaddr*) 0 )
   hs->listen4_fd = -1;
  else
   hs->listen4_fd = initialize_listen_socket( sa4P );
  /*如果没有得到任何监听套接字，则释放前面分配的空间并退出程序*/
  if ( hs->listen4_fd == -1 && hs->listen6_fd == -1 )
   {
   free_httpd_server( hs );
```

```
 return (httpd_server*) 0;
 }
/*初始化 MIME Type，即该资源的媒体类型，浏览器中显示的内容有 HTML、XML、GIF、
Flash、video、vrml 等等，浏览器通过 MIME Type 来区分它们*/
init_mime();

/*Done initializing.*/
if ( hs->binding_hostname == (char*) 0 )
 syslog(
     LOG_NOTICE, "%.80s starting on port %d", SERVER_SOFTWARE,
     (int) hs->port );
else
 syslog(
     LOG_NOTICE, "%.80s starting on %.80s, port %d", SERVER_SOFTWARE,
     /*将 httpd_sockaddr 结构体转化为数字加点组成的 IP 地址字符串*/
     httpd_ntoa( hs->listen4_fd != -1 ? sa4P : sa6P ),
     (int) hs->port );
return hs;
}
```

# 18.2　Thttpd 编译和 HTML 页面测试

第 17 章已经介绍了 BOA 的编译，与 BOA 或其他 Web 服务器都类似的是 Thttpd 也有配置文件 thttpd.conf，该文件在 contrib/redhat-rpm 目录下。

## 18.2.1　配置文件介绍

配置文件对于任何 Web 服务器都是非常重要的，Tomcat 是在 Windows 上开发 Web 程序常用的服务器。配置 Tomcat 时有个格式固定的 server.xml 文件，在其中填写对应的内容。Thttpd 也有配置文件 thttpd.conf，其配置也是一样的，和第 17 章 BOA 的配置文件 boa.conf 类似。下面先给出配置文件，在编译的时候如果出现错误，首先对照配置文件进行查找。

配置文件 Thttpd.conf 比较简单，主要指定 HTML 文件路径、日志文件、PID 文件等。配置文件 Thttpd.conf 中注释的部分为默认配置的部分，包括端口号、主机 IP、字符编码格式等信息，具体内容如下：

```
dir=/home/httpd/html                              #html 文件路径
chroot
user=httpd# default = nobody                      #用户名
logfile=/var/log/thttpd.log                       #指定日志文件
pidfile=/var/run/thttpd.pid                       #指定 PID 文件
# This section  documents  defaults in effect
# port=80                                         #默认的端口号
# nosymlink# default = !chroot
# novhost
# nocgipat                                        #不指定 CGI 程序的路径
# nothrottles
# host=0.0.0.0                                    #不设置指本机 IP
```

```
# charset=iso-8859-1                                    #默认的编码方式
```

修改该宏定义后，再执行 make 进行编译，在 src 目录下生成 thttpd 可执行程序。

## 18.2.2　Thttpd 编译

编译 Thttpd 的过程与前面介绍的编译方法基本类似，这里按照：编译→遇到问题→修正后重新编译的顺序进行。下面介绍 Thttpd 编译的详细过程。

（1）准备 Thttpd 的源代码。这里使用的源文件为 thttpd-2.25b.tar.gz，读者可以到网上下载更新的版本。

（2）创建安装目录。在编译安装源代码时，创建自己的安装目录，也可以按照默认的方式安装。笔者认为创建一个安装目录比较合适，安装完成后可以很快知道安装目录下生成哪些工具和哪些库等文件。

```
# mkdir /usr/local/thttpd_x86
```

（3）解压源代码和编译。解压源码后，进入代码目录使用 configure 命令生成 Makefile 文件。然后进行编译，执行 make install 进行安装。

```
# ./configure --prefix=/usr/local/thttpd x86
# make
# make install
```

在执行 make install 时，会提示子目录安装有问题，不用理会这些错误程序依然可以执行，查看安装目录是否生成所需要的文件。在安装目录/usr/local/thttpd_x86 下面会生成 3 个文件：sbin 用于放工具文件，man 用于放手册，www 用于放 CGI 和 HTML 文件。下面给出如何解决这些错误的方法。

- 错误 1：缺少 www 组用户，可以使用 adduser www 增加一个组用户，通过# cat /etc/group 命令显示该文件下多了一个 www 组用户。

```
# adduser www
# cat/etc/group
```

- 错误 2：没有使用手册 man1 的路径，这在以前的编译和移植过程中也遇到类似的错误，手动在安装目录下创建一个存放手册文件的路径。

```
# mkdir/usr/local/thttpd_x86/man/man1
```

## 18.2.3　运行和测试 Thttpd

下面运行生成的 Web 服务器，并通过 HTML 进行测试。本节将介绍 3 部分内容：编写测试主页，运行 Web 服务器，通过 HTML 进行测试。

### 1．编写测试主页index.html

编写 HTML 非常简单，如果读者不想自己动手写代码，直接打开浏览器，找一个自己喜欢的风格的主页，下载后进行修改。如果会使用 Dreamweaver 更好，如不会使用直接用

UEdit 或者写字板也可以进行编辑。

**注意：** 修改 HTML 时，直接查找要修改部分的关键字进行修改，查找<title></title>，<head></head>，<body></body>等直接进行内容修改。

### 2. 启动Web服务器

运行 Thttpd 服务器时要指定配置文件。配置文件 thttpd.conf 在 contrib/redhat-rpm 目录下。可以将配置文件复制到 sbin 目录下，也可以在运行 Web 服务器时，指定配置文件的绝对路径。

```
#./thttpd -C thttpd.conf
```

或者

```
#./thttpd -C /usr/local/thttpd x86/thttpd-2.25b/contrib/redhat-rpm/
thttpd.conf
```

-C 表示指定配置文件，如果没有带上参数，系统会自动提示读者带上参数，并且给出很多参数的含义。运行的时候会提示没有指定用户 httpd，回头查看下配置文件 user=httpd# default = nobody，即程序中默认的 user 为 nobody，目前的 user 指定为 httpd，应该使用 adduser 命令添加一个 user 为 httpd。

```
./thttpd: unknown user - 'httpd'
```

解决上面的错误，同样是增加一个 user 名为 httpd。当然也可以修改配置文件，在配置文件中将 user 修改为默认的 user，或者文件/etc/passwd 中存在的 user。

```
# adduser httpd
```

**注意：** 这里暂时不改配置文件，而在实际项目开发中一般是先部署整个网站的框架，即先部署网站必需的文件夹和相关文件，然后再修改配置文件，让配置文件适应实际的部署。

### 3. 测试HTML

在测试主机的浏览器地址栏中输入 http://192.168.1.123（服务器的 IP 地址），通过 HTML 主页访问服务器测试。前面在做 BOA 测试的时候，虚拟机和主机采用 NAT 的方式连接，所以虚拟机和主机不在一个 IP 段也能进行访问。使用 NAT 方式时，不插上网线，也能通过主机对虚拟机进行测试。这次虚拟机和主机采用的是网桥方式，采用这种方式是为了直接将虚拟机和开发板连接起来。测试的时候一定要注意记得插上网线。

在主机的浏览器地址栏中输入 http://192.168.1.123，读者测试时改成自己的虚拟机 IP 地址。如果成功则在主机浏览器中看到服务器的主页。如果出错则有两种常见错误。

- ❑ 错误 1：回车运行得到无法显示网页。原因是前面修改了 index.html，而没有对它进行部署。

解决方法：正确部署文件 index.html。首先查看配置文件指定 HTML 路径的设置 dir=/home/httpd/html。可以将 index.html 部署在该目录下，也可以将该配置修改为 dir=

/usr/local/thttpd_x86/www。

```
# cp index.html /home/httpd/html
```

或者

```
# cp index.html /usr/local/thttpd_x86/www
同 时 修 改 配 置 文 件  thttpd.conf ， 将  dir=/home/httpd/html  改 为 ： dir=
/usr/local/thttpd_x86/www
```

**注意：**服务器修正问题后，保持配置和文件，客户端只要刷新就能获得更新后的信息。

- 错误 2：测试的时候可能会遇到其他问题，导致无法正确显示网页。这时应该查看日志文件/var/log/thttpd.log。从查看配置文件获得日志文件的路径中使用 cat 命令查看日志文件的记录。

```
# cat/var/log/thttpd.log
```

下面是笔者测试出错后在日志中查看到的一条记录，同时在主机的浏览器中得到禁止访问，如图 18.1 所示。

```
192.168.1.199 - - [11/Apr/2010:14:45:45 +0000] "GET / HTTP/1.1" 403 0 ""
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
1.1.4322; .NET CLR 2.0.50727)"
```

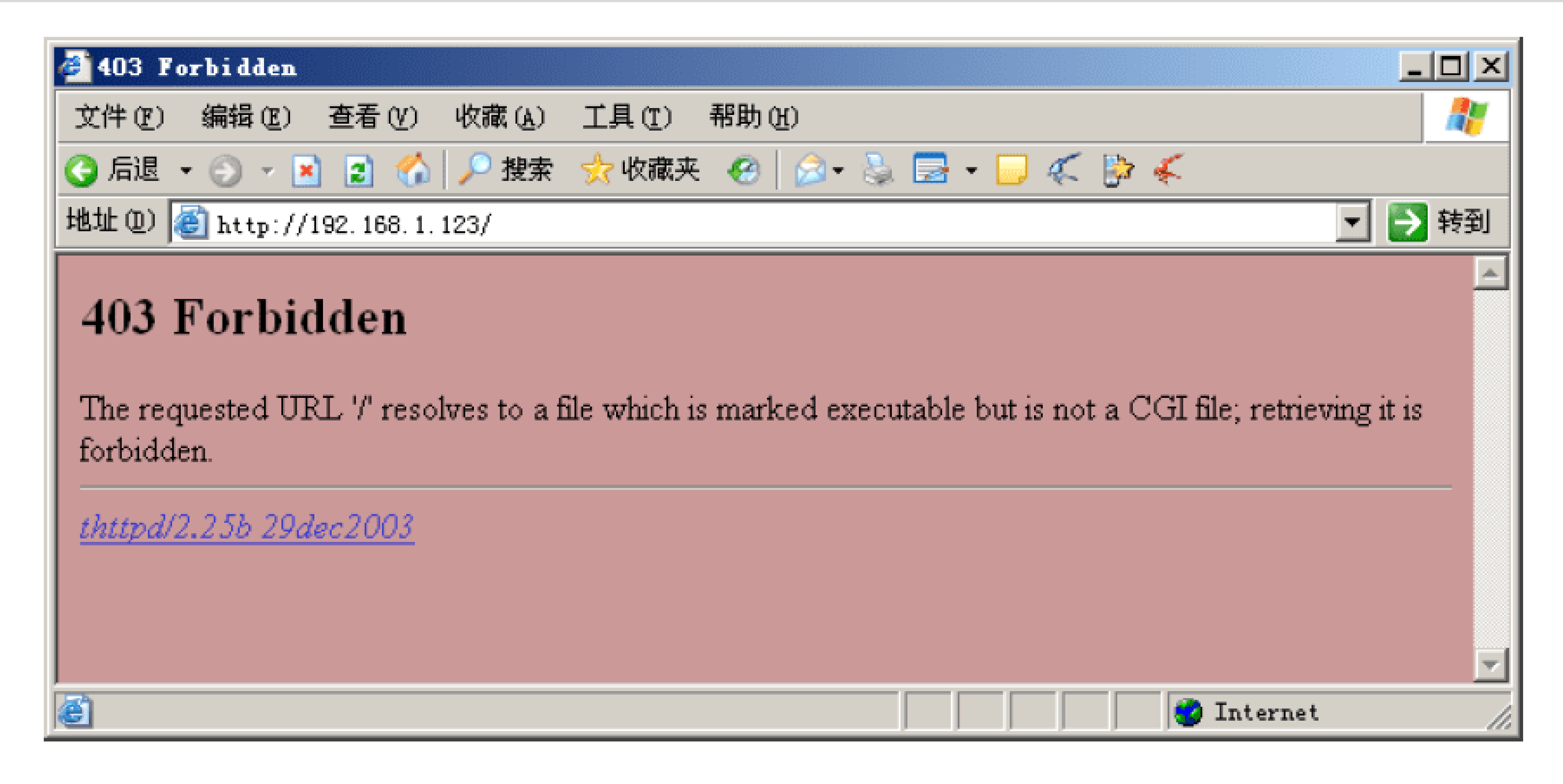



图 18.1　主机浏览器中的显示

错误分析：返回码为“403”：说明服务器已经收到了浏览器的请求，同时出于安全考虑禁止浏览器进行访问。查看 index.html 的执行属性，发现其属性为：

```
-rwxrwxrwx 1 root root 422 04-11 21:44 index.html
```

解决方法：修改文件的权限，使其属性为只读，修改命令如下：

```
# chmod 444 index.html
# ls -l
-r--r--r-- 1 root root 422 04-11 21:44 index.html
```

修改权限后，能够正确浏览测试页面 index.html，如图 18.2 所示。

- 错误 3：网页中可能会出现乱码。

解决方法：修改配置文件的中字符编码方案，将 charset 设置为 utf-8。笔者做过试验

将 charset 设置为 gbk 和 gb2312 仍不能解决中文乱码问题。设置为 utf-8 后显示中文正常，如图 18.3 所示。

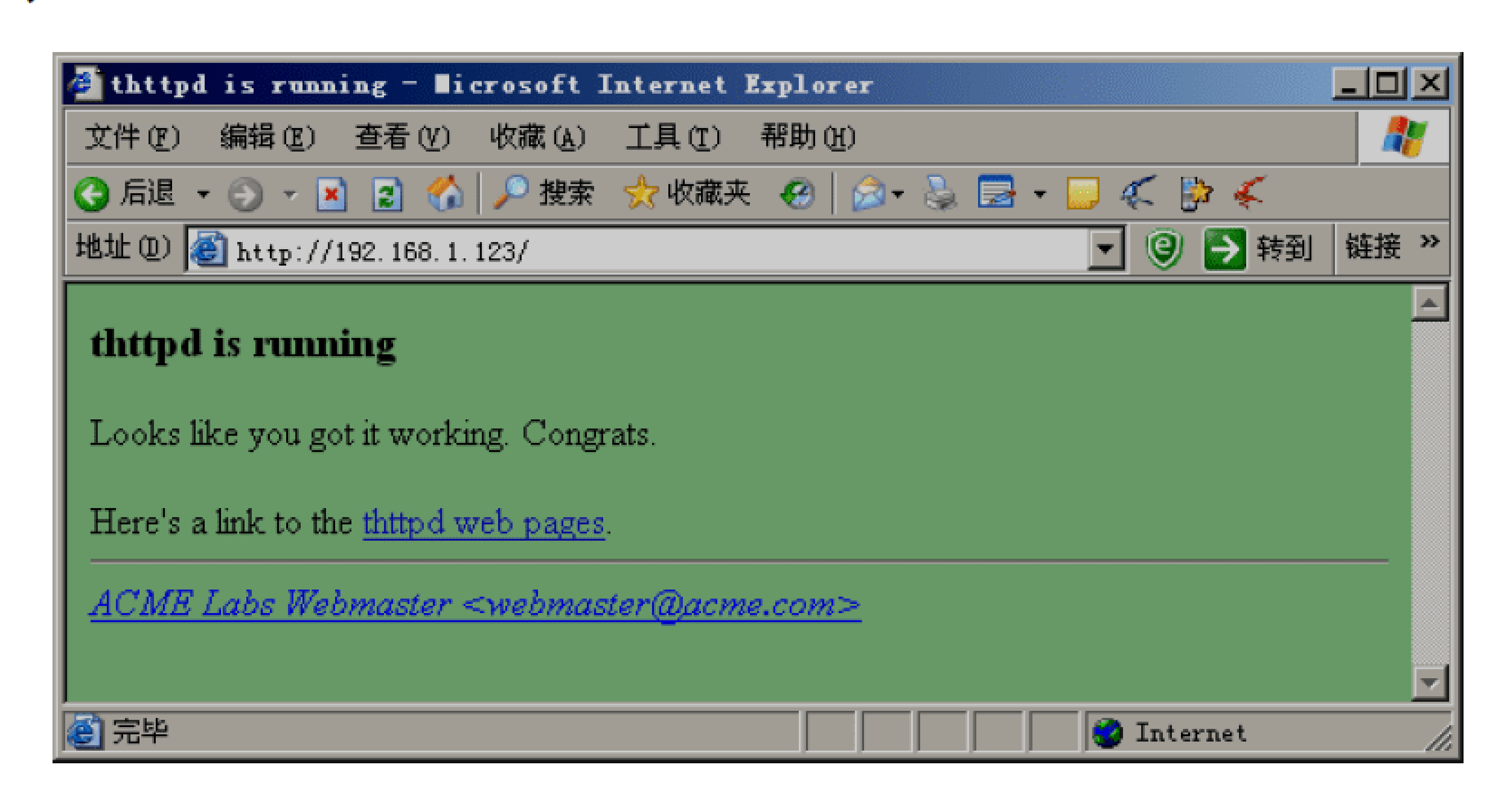



图 18.2　正确的测试主页

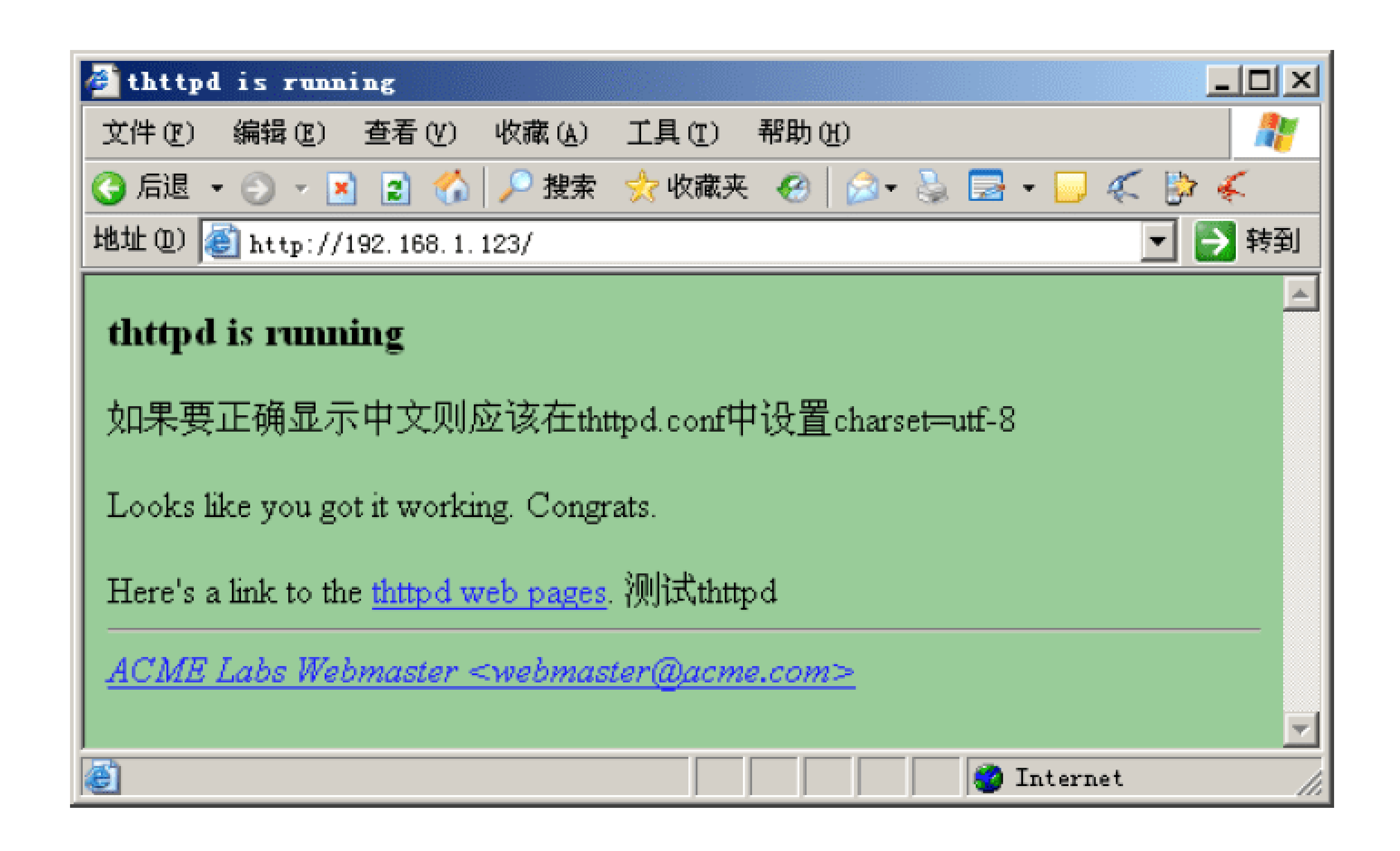



图 18.3　正确显示中文

# 18.3　CGI 脚本测试

下面测试 CGI 脚本程序。对 CGI 程序进行测试时，首先应该修改配置文件，指定 CGI 程序的路径。CGI 脚本的测试也包括 3 个部分：编写测试代码、编译测试代码、执行测试。

## 18.3.1　编写测试代码

CGI 的文件应该放在目录/home/httpd/html/cgi-bin 下，同时修改 thttpd.conf 修改 nocgipat 为 cgipat。

```
cgipat=/cgi-bin
```

在 /home/httpd/html/cgi-bin 目录下编写 hello.c 文件，该测试文件内容为打印

“Hello,World.” 。测试文件的代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
	printf("Content-type: text/html\n\n");
	printf("<html>\n");
	printf("<head><title>CGI Test For Thttpd</title></head>\n");
	printf("<body>\n");
	printf("<h1>Hello,world.</h1>\n");
	printf("<body>\n");
	printf("</html>\n");
	exit(0);
}
```

### 18.3.2　编译测试程序

将 hello.c 文件编译生成 hello.cgi 文件。编译命令如下：

```
# gcc -o hello hello.c
```

编译生成 hello 后，如果该文件不是在目录/home/httpd/html/cgi-bin 下，则将 hello 文件复制到该目录下。如果编译的文件名为 hello.cgi，在测试的过程中，就会出现下载该文件。编译好的 CGI 文件同样需要将其权限修改为只读。

```
# chmod 444 hello
```

### 18.3.3　测试 CGI 脚本

打开客户端的浏览器，在浏览器中输入下面地址进行访问。

```
http://192.168.1.123/cgi-bin/hello
```

hello 的权限也要设置为可读方式，否则也无法正确显示。

## 18.4　Thttpd 交叉编译与移植

本节将介绍如何在嵌入式产品中应用 Thttpd，在嵌入式产品中使用 Thttpd 需要对其进行交叉编译、配置、编写 HTML 页面、编写 CGI、部署上述文件到相应的目录。

### 18.4.1　交叉编译 Thttpd

进入 Thttpd 源码的解压目录对 Thttpd 进行交叉编译，这里使用的交叉编译器为 arm-linux-gcc-4.3.2。编译的过程如下：

（1）在上位机中调试的时候已经通过 configure 命令生成了 Makefile 文件，这里只需要对生成的 Makefile 文件进行修改。在 Makefile 文件中将 gcc 改为 arm-linux-gcc，将安装目录指定为/usr/local/thttpd_arm，同时也建立这样一个目录。

```
# mkdir /usr/local/thttpd_arm
#vi Makefile
CC=arm-linux-gcc
prefix=/usr/local/thttpd arm
# make clean                    //之前进行了 X86 平台的编译，清除编译生成的文件
# make
```

（2）编译完成后进行 make install 安装，安装之前需进行一些设置，这与在 X86 平台上编译安装的过程相同。

```
# mkdir /usr/local/thttpd arm/man/man1
# make install
```

## 18.4.2　交叉编译 CGI 程序

测试程序包括测试的 HTML 页面和 CGI 程序，HTML 和 CGI 程序，使用在上位机中测试的程序。这里只需要对 CGI 程序进行交叉编译即可以使用在 ARM 平台上。

```
# arm-linux-gcc -o hello hello.c
```

同样，执行完成后对生成的 hello 文件查看其文件信息，同时修改其权限。

```
# chmod 444 hello
```

## 18.4.3　移植 Thttpd

在开发板上部署 Web 服务器。部署的文件包括：部署配置信息 thttpd.conf，部署服务器程序 Thttpd，部署访问页面和 CGI 程序及相关目录，移植 Thttpd 需要的库。

（1）部署配置信息 thttpd.conf，复制上位机/etc/thttpd.conf 到开发板的/etc 目录下。配置文件的信息如下：

```
dir=/home/httpd/html
chroot
user=httpd# default = nobody
logfile=/var/log/thttpd.log
pidfile=/var/run/thttpd.pid
# This section _documents_ defaults in effect
# port=80
# nosymlink# default = !chroot
# novhost
cgipat=/cgi-bin/*
# nothrottles
# host=0.0.0.0
charset=utf-8
```

（2）部署服务器程序 Thttpd，复制上位机/usr/local/thttpd_arm/sbin 目录下的 thttpd 到开发板/usr/sbin 目录下。

（3）在开发板上也建立目录/home/httpd/html，同时将 index.html 部署在该目录下，在该目录下建立目录 cgi-bin，将交叉编译好的 CGI 程序放置在该目录下。

（4）复制 Thttpd 依赖的库文件，可以对 X86 版的 thttpd 使用 ldd 命令查看其依赖的库文件。从交叉编译路径下复制这些库到开发板的 lib 库下。

```
# ldd thttpd
        linux-gate.so.1 =>  (0x00b96000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x45231000)
        libc.so.6 => /lib/libc.so.6 (0x44588000)
        /lib/ld-linux.so.2 (0x43bb9000)
```

查看开发板/lib 目录，存在上述文件。依赖库文件就不需要复制。

## 18.4.4　测试

移植到开发板上时，主要是配置内核，添加用户，修改权限等问题。可能读者在开发板上测试时还可能会发生其他问题。如果遇到其他问题就查看日志/var/log/messages。

（1）内核支持 IPv6，在内核中添加对 IPv6 的支持，如图 18.4 所示。

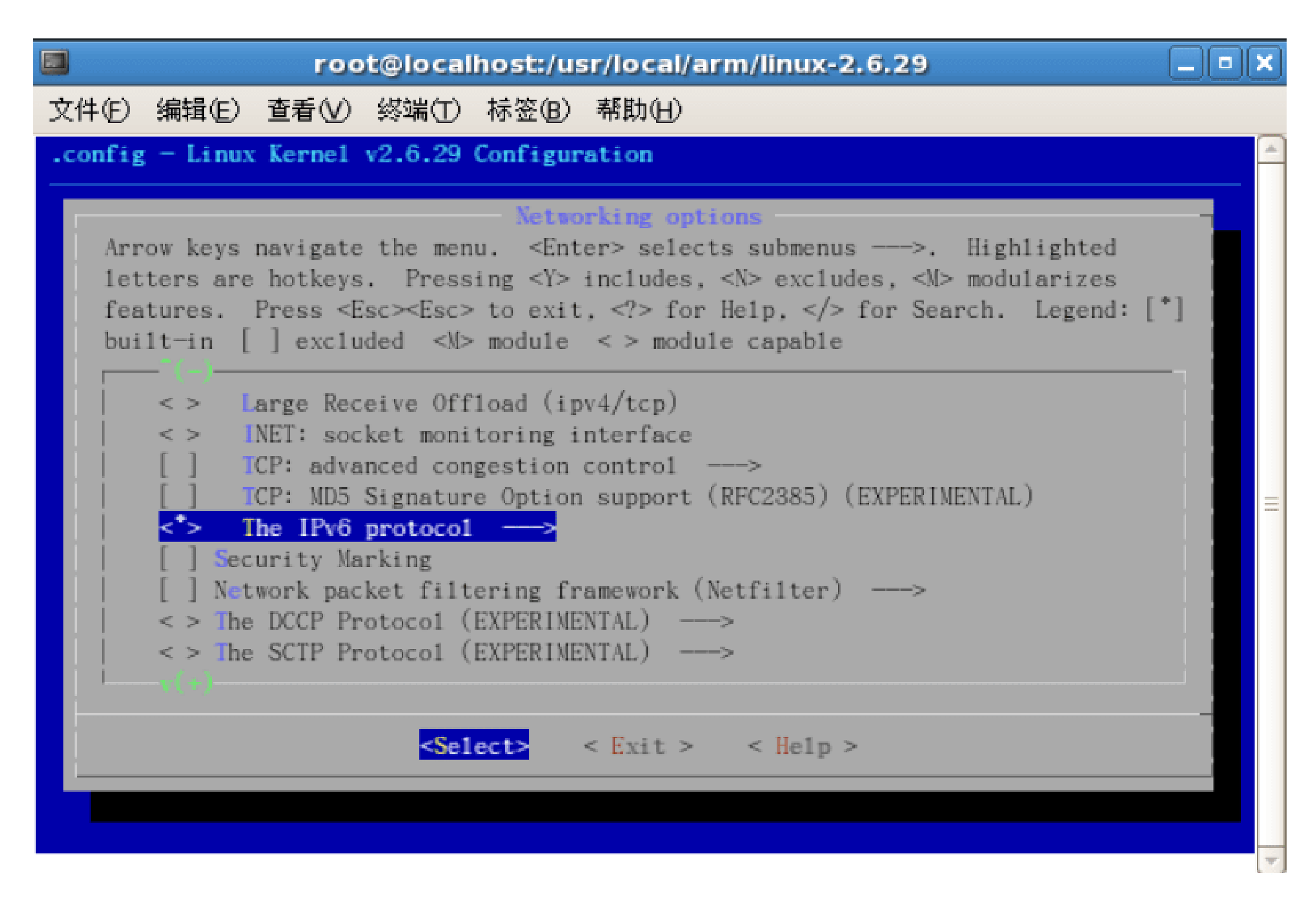



图 18.4　内核支持 IPv6

（2）修改 thttpd 的执行权限。

```
# chmod 777 thttpd
```

（3）增加用户 httpd。

```
# adduser httpd
```

**注意：** 移植到 mini2440 上时，自带的文件系统默认启动 boa Web 服务，去掉开发板自带的文件系统的/etc/init.d/rcS 中运行的 boa 代码，然后重新启动。

（4）启动 Web 服务器，并通过主机进行测试。测试结果和主机上一样。

# 18.5　Thttpd 与嵌入式数据库结合

在前面章节中，已经介绍了嵌入式 Web 服务器 BOA 与 SQLite 在项目中结合使用的情况。本节将以实例介绍通过 Thttpd Web 服务 CGI 接口维护和管理嵌入式数据库。

## 18.5.1　通过 CGI 程序访问 SQLite

SQLite 提供了 C 语言访问的接口，通过采用 C 语言程序访问数据库，将该访问数据库的操作编译成 CGI 程序部署在 Thttpd 的 CGI 路径下，远程维护人员通过调用此 CGI 程序就能实现远程维护数据库的目的。

下面是例子程序，其代码主要分为两部分：一部分是通过调用 C 接口对数据库进行创建、修改等维护工作；另一部分是通过 HTML 页面将结果返回给远程访问者。是在第 17 章代码的基础上稍加改动。具体代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

int main( void )
{
    sqlite3 *db=NULL;
    char *zErrMsg = 0;
    int rc;
    int i=0;
    //通过 CGI 将结果返回给远程操作者
    printf("<%@ page contentType=\"text/html; charset=utf-8\"%>");
                                    //设置编码方案，解决中文乱码
    printf("<html>\n");
     printf("<head><title>CGI Output</title></head>\n");
     printf("<body>\n");
     printf("<h1>Access SQLite Database by CGI of Thttpd</h1>\n");
    rc = sqlite3 open("test.db", &db);
                                    //打开指定的数据库文件，如果不存在将创建
                                      一个同名的数据库文件
    if( rc )
    {
        fprintf(stderr, "Can't open database: %s\n", sqlite3 errmsg(db));
        sqlite3 close(db);
        exit(1);
    }
    else
        printf("opened database test.db successfully!\n");

    //创建一个表，如果该表存在，则不创建，并给出提示信息，存储在 zErrMsg 中
    char *sql = "create table student(id, name , sex, age);" ;
    //执行 SQL 语句
    sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );

    //插入数据
    sql = "insert into student values(1, 'Jack', 'M', 20);" ;
```

```
//执行 SQL 语句
sqlite3_exec( db , sql , 0 , 0 , &zErrMsg );
//插入数据
sql = "insert into student values(2, 'Tom', 'M', 21);" ;
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
//插入数据
sql = "insert into student values(3, 'Mary', 'W', 19);" ;
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );
//在 HTML 中显示操作的内容
printf("在表 student 中插入数据项(1, 'Jack', 'M', 20);");
printf("<p>\n");
printf("在表 student 中插入数据项(2, 'Tom', 'M', 21);");
printf("<p>\n");
printf("在表 student 中插入数据项(3, 'Mary', 'W', 19);");
printf("<p>\n");

//查询结果
sql = "select * from student;" ;
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

//查询数据
/*
int sqlite3_get_table(sqlite3*, const char *sql,char***result , int
*nrow , int *ncolumn ,char **errmsg );
result 中是以数组的形式存放所查询的数据，首先是表名，再是数据。
nrow ,ncolumn 分别为查询语句返回的结果集的行数，列数，没有查到结果时返回 0
*/
int nrow = 0, ncolumn = 0;
char **fristResult;                    //二维数组存放结果

sql = "select * from student;" ;
printf("\n");
sqlite3_get_table(db, sql, &fristResult, &nrow, &ncolumn, &zErrMsg);

//打印查询结果
printf( "row: %d  col: %d \n" , nrow , ncolumn );
printf( "\n 更新前数据库的数据: \n" );

for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ ){
if (i%4==0) printf("<p>\n");
printf( "%s\t", i , fristResult[i] );
}

//释放 fristResult 的内存空间
sqlite3_free_table( fristResult );

sql = "update student set age = 24 where age = 20;" ;
printf("\n");
sqlite3_exec( db , sql, 0 , 0 , &zErrMsg );

nrow = 0;
ncolumn = 0;
char **secondResult;                   //二维数组存放结果
//查询数据
sql = "select * from student;" ;
sqlite3_get_table( db , sql , &secondResult , &nrow , &ncolumn ,
&zErrMsg );
 printf("<p>\n");
```

```
     //打印查询结果
     printf( "row:%d column=%d \n" , nrow , ncolumn );
     printf( "\n 更新后的数据库的结果: \n" );

     for( i=0 ; i<( nrow + 1 ) * ncolumn ; i++ ) {
    if (i%4==0) printf("<p>\n");
    printf( "%s\t", i , fristResult[i] );
    }

     printf("<body>\n");
     printf("</html>\n");
    //释放 secondResult 的内存空间
    sqlite3 free table( secondResult );

    printf("\n");
    sqlite3_close(db);                    //关闭数据库
    return 0;
}
```

## 18.5.2　编译和测试

与 BOA 中大致相同，编译 SQLite 程序时，需要 SQLite 接口的头文件和库文件支持。测试前对 CGI 进行编译和部署。

### 1. CGI程序的编译和部署

将上述代码起名为 sqlite.c 放在 Thttpd 服务器的 CGI 路径下，查看 Thttpd 的配置文件/etc/thttpd.conf，对于本机的路径，CGI 文件的路径设置为/home/httpd/html/cgi-bin。编译时需指定依赖的头文件和库文件，具体的编译命令如下：

```
#gcc  -o  sqlite  -I  /usr/local/sqlite_x86/include  -L  /usr/local/
sqlite_x86/lib sqlite.c -lsqlite3 -static -lpthread
```

/usr/local/sqlite_x86 是本机 SQLite 的安装目录。编译完成后在/home/httpd/html/cgi-bin 目录下生成了 sqlite 文件，修改其属性为只读，同时也完成了部署。

```
#chmod 444 sqlite
```

**注意：**对于 Thttpd 的 CGI 程序，一般命名不带后缀.cgi。命名采用带后缀名的 CGI 程序时，测试的时候，将会得到通过网页下载该文件，而并非是在网页中显示。

### 2. 测试sqlite

启动 Thttpd Web 服务，在 Windows 的浏览器中输入 http://192.168.1.123/cgi-bin/sqlite，来访问虚拟机（192.168.1.123 是虚拟机的 IP 地址）下的 CGI 程序。

```
#./thttpd -C/etc/thttpd.conf                    //启动 Thttpd Web 服务
```

**注意：**前面已经介绍 Thttpd 的移植过程，对于和数据库结合移植的过程留给读者完成。

## 18.6　小　　结

除了 Thttpd、Boa 外，还有很多嵌入式 Web 服务器，其工作流程及移植过程基本与 Boa、Thttpd 类似。读者可以试着去将前面介绍的技术与之结合起来应用在实际的开发中，如和 GUI 数据库的结合应用到自己的嵌入式项目中。

# 第 19 章　JVM 及其移植

JVM 即 Java 虚拟机（Java Virtual Machine），是虚拟出来的计算机，在实际的计算上通过软件模拟出各种硬件的功能，如处理器、堆栈、寄存器、指令系统等。JVM 屏蔽了具体平台的信息，使得 Java 程序能运行在各种安装了 JVM 的平台上。本章内容主要包括 JVM 原理、作用、移植及编程。在介绍移植实例前还会对 Java 程序进行简单分析。

## 19.1　JVM 介绍

JVM（Java 虚拟机）一种用于计算设备的规范，可采用不同的方式（软件或硬件）实现。编译虚拟机的指令集与编译微处理器的指令集基本类似。Java 虚拟机的组成部分包括一套字节码指令集、一组寄存器、一个栈、一个垃圾回收堆、一个存储方法域和一个执行引擎。Java 虚拟机（JVM）是可运行 Java 代码的虚拟计算机。按照 JVM 规格描述将解释器移植到特定的计算机上，就能保证经过编译的任意 Java 代码都能够正确运行在该系统上。Java 虚拟机是一个虚拟出来的机器，通过在实际计算机上采用软件模拟实现。Java 虚拟机模拟的硬件包括处理器、堆栈、寄存器、指令系统等。

### 19.1.1　JVM 原理

Java 语言的特点就是平台无关性，这一特性主要是通过 Java 虚拟机来实现。其他高级语言如果要在不同的平台上运行，至少应该重新编译成不同的目标代码。而使用 Java 虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 程序经过编译后运行在 Java 虚拟机上，屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在不同平台（安装了 JVM）上不加修改地运行。Java 虚拟机在执行字节码时，将字节码解释成对应平台上的机器指令进行执行。下面给出 JVM 原理图，如图 19.1 所示，后面将会对其实现细节进行介绍。

JVM 生命周期开始于运行 Java 程序，消亡于 Java 程序的关闭退出。Java 虚拟机实例通过调用任意某个初始类的 main()方法运行一个 Java 程序。只要还有任何非守护线程在运行，那么这个 Java 程序也在继续运行（虚拟机仍然存活）。

Sun 公司提供了 3 种运行在小型设备操作系统上的 JVM，分别为 CVM、KVM 和 Card VM，这 3 种 JVM 有不同的应用。

- CVM：应用于瘦客户端，如数字机顶盒、车载电子系统等；
- KVM：应用于电池供电的手持移动设备，如移动电话、PDA 等；
- Card VM：应用于智能卡（Smart Card）系统。

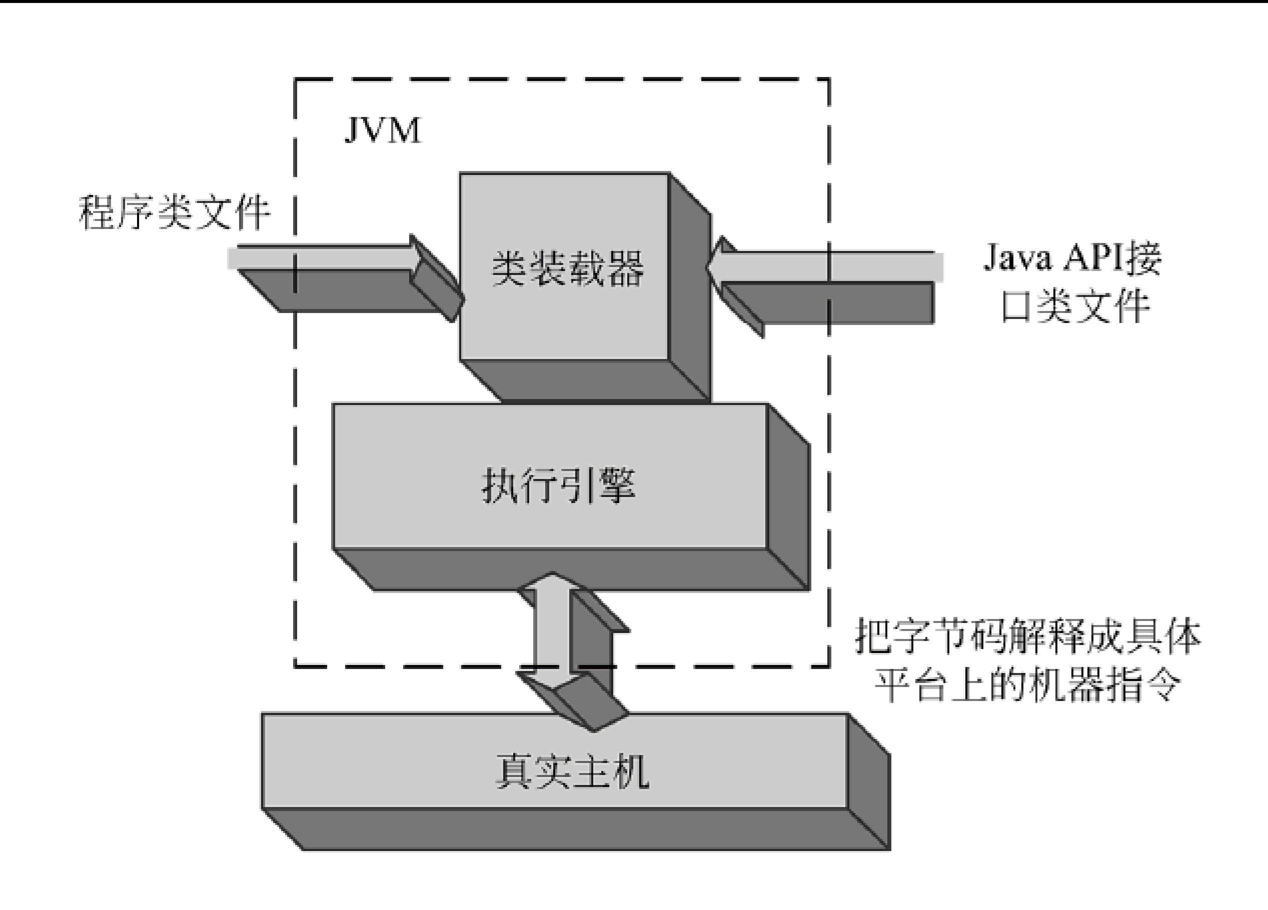



图 19.1　JVM 原理图

CVM、KVM 和 Card VM 3 者适用的硬件资源是由高到低的，根据不同的硬件选择不同的虚拟机。KVM 完成的功能是 CVM 完成功能的子集。CVM 允许设备将 Java 线程映射为本地线程，完成垃圾收集、Java 同步等。在存储系统方面，表现为精确、高效的垃圾收集，虚拟机与存储系统分离；在可移植性方面，CVM 使用 C 语言，实现快速、安全地移植；在本地线程方面，CVM 支持线程抢占。

而 KVM 的最大特点是小而高效，只需要几万字节的存储空间就可以运行。虚拟机和类库大小约 50～80KB，具有较高的可移植性和可扩展性，垃圾收集独立于系统，支持多线程。在后面的介绍中，多以 KVM 为例讲解。

## 19.1.2　JVM 支持的数据类型

Java 虚拟机不仅支持 Java 语言的基本数据类型，而且支持其他数据类型。Java 支持的基本数据类型类似 C 语言中的 int、long 等，支持的数据结构如表 19.1 所示。

表 19.1　JVM支持的数据结构

| 分　类 | 数 据 类 型 | 说　明 |
|---|---|---|
| 基本的数据类型 | byte | 1 字节有符号整数的补码 |
| | short | 2 字节有符号整数的补码 |
| | int | 4 字节有符号整数的补码 |
| | long | 8 字节有符号整数的补码 |
| | float | 4 字节 IEEE754 单精度浮点数 |
| | double | 8 字节 IEEE754 双精度浮点数 |
| | char | 2 字节无符号 Unicode 字符 |
| 其他数据类型 | object | 4 字节，对一个 Javaobject（对象）的引用 |
| | returnAddress | 4 字节，用于 jsr/ret/jsr-w/ret-w 指令 |

应用程序为了能够编译成为 Java 虚拟机的字节码且正确执行，就必须遵守 JVM 支持的类型规定。Java 虚拟机不执行违反了类型规定的字节码程序。从 Java 虚拟机支持的数据

类型看，Java 对数据类型的内部格式进行了严格规定，这样使得各种 Java 虚拟机的实现对数据的解释是一致的，从而保证了 Java 的与平台无关性和可移植性。

### 19.1.3　JVM 指令系统

JVM 指令系统与其他计算机的指令系统类似。Java 指令也是由操作码和操作数两部分组成。操作码是 8 位二进制数，操作数紧跟在操作码后面，其长度根据需要而不同。操作码指定一条指令操作的功能，如 iload 表示从存储器中装载一个整型数，anewarray 表示给一个新数组分配空间，iand 表示两个整数的“与”操作，ret 用于流程控制，表示从某方法的调用中返回。当长度大于 8 位时，操作数被拆分为两个以上字节进行存放。JVM 采用了“big endian”的编码方式，即低位 bits 存放在高字节，高位 bits 存放在低字节。

Java 指令系统是专门为 Java 语言的实现而设计的，还包括用于调用方法和监视多线程系统的指令。Java 的操作码长度为 8 位，使得 JVM 最多有 256 种指令，目前已使用了其中的 160 多种操作码。

### 19.1.4　JVM 寄存器

所有的 CPU 均包含用于保存系统状态和处理器所需信息的寄存器组。如果虚拟机定义较多的寄存器，便可以从寄存器中得到更多的信息而不必对栈或内存进行访问，这有利于提高系统的运行速度。然而，如果虚拟机中的寄存器比实际 CPU 的寄存器还多，那么在实现虚拟机时就会占用处理器大量的时间来用常规存储器模拟寄存器，这反而会降低虚拟机的效率。因此，JVM 只设置了以下 4 个最为常用的寄存器：

- pc：程序计数器。
- optop：操作数栈顶指针。
- frame：当前执行环境指针。
- vars：指向当前执行环境中第一个局部变量的指针。

**说明：**所有寄存器均为 32 位。pc 用于记录程序的执行、optop、frame 和 vars 用于记录指向 Java 栈区的指针。

### 19.1.5　JVM 栈结构

作为基于栈结构的计算机，Java 栈是 JVM 存储信息的主要手段。当 JVM 获得一个 Java 字节码应用程序后，便为该代码中一个类的每一个方法创建一个栈框架，以保存该方法的状态信息。每个栈框架包括 3 类信息：局部变量、执行环境和操作数栈。在线程的执行栈中分配帧，下面为 KVM 帧结构体（下面例子都以 JVM 中的 KVM 为例介绍）：

```
struct frameStruct {
    FRAME    previousFp;          //保存前向帧指针
    BYTE*    previousIp;          //保存前向的程序计数器
    cell*    previousSp;          //保存前向的栈指针
```

```
    METHOD   thisMethod;          //指向当前执行的方法
    STACK    stack;               //stackStruct 类型的指针
    OBJECT   syncObject;          //objectStruct 类型指针
};
```

在必要的时候，JVM stack 可以动态增加和减小。每个线程都拥有一个私有的 JVM stack，这个堆栈与线程一同创建。JVM 栈和 C 语言的栈结构相似。由于 JVM 的帧可以存放堆上，所以 JVM stack 可以是不连续的。JVM 的设计者让程序员可以控制初始栈的大小，并控制栈的最大、最小值。下面通过 KVM 栈结构体：

```
struct stackStruct {
    STACK    next;           //指向下一个 stackStruct 指针，即该结构为链表
    short    size;
    short    xxunusedxx;     /*must be multiple of 4 on all platforms*/
    cell     cells[STACKCHUNKSIZE];
};
```

对于栈的操作一般有 push、pop 等操作，下面为 KVM 栈操作函数。

### 1．PushFrame()函数操作

函数 pushFrame()为当前执行线程创建一个新的栈帧。在分配帧前，所有的寄存器都必须被正确地初始化。KVM 与其他 JVM 不同，它不为局部函数调用创建栈帧。

```
void pushFrame(METHOD thisMethod)
{
    int thisFrameSize = thisMethod->frameSize;
    int thisArgCount = thisMethod->argCount;
    int thisLocalCount = thisFrameSize - thisArgCount;

    STACK stack = getFP() ? getFP()->stack : CurrentThread->stack;
    int thisMethodHeight = thisLocalCount + thisMethod->u.java.maxStack +
                           SIZEOF_FRAME + RESERVEDFORNATIVE;
    FRAME newFrame;
    int i;
    cell* prev_sp = getSP() - thisArgCount; /* Very volatile! */

    /*检查当前的栈块中是否有足够的空间*/
    if (getSP() - stack->cells + thisMethodHeight >= stack->size) {
        /*没有足够的栈块空间时，需要重新创建一个新的栈块或者重新使用已分配的一个栈块*/
        STACK newstack;                      //重新创建新栈块
        thisMethodHeight += thisArgCount;
        /*检查是否可以重新使用已分配的一个栈块*/
        if (stack->next && thisMethodHeight > stack->next->size) {
            stack->next = NULL;
        }
        /*如果下一栈为空，则需要需要重新创建栈块*/
        if (stack->next == NULL) {
            int size = thisMethodHeight > STACKCHUNKSIZE ? thisMethodHeight :
            STACKCHUNKSIZE;
            int stacksize = sizeof(struct stackStruct) / CELL +  (size -
            STACKCHUNKSIZE);
            START_TEMPORARY_ROOTS
                DECLARE_TEMPORARY_ROOT(STACK, stackX, stack);
                newstack = (STACK)mallocHeapObject(stacksize,
                GCT_EXECSTACK);
                stack = stackX;
```

```
                prev_sp = getSP() - thisArgCount;
            END_TEMPORARY_ROOTS
            if (newstack == NULL) {
                THROW(StackOverflowObject);
            }
        } else {
            /*可以使用一个存在的、未使用的栈块*/
            newstack = stack->next;
        }
        /*将新分配的栈加入到栈链表中*/
        for (i = 0; i < thisArgCount; i++) {
            newstack->cells[i] = prev_sp[i + 1];
        }
        setLP(newstack->cells);
        newFrame = (FRAME)(getLP() + thisFrameSize);
        newFrame->stack = newstack;
    } else {

        ASSERTING NO ALLOCATION
            /*设置开始指针指向执行栈中的局部变量*/
            setLP(prev sp + 1);
            /*为局部变量增加空间，并初始化新的帧指针*/
            newFrame = (FRAME)(getSP() + thisLocalCount + 1);
            newFrame->stack = stack;
        END ASSERTING NO ALLOCATION
    }
    /*填充栈帧中的 remaining 域*/
    ASSERTING_NO_ALLOCATION
        /*Initialize info needed for popping the stack frame later on*/
        newFrame->previousSp = prev_sp;
        newFrame->previousIp = getIP();
        newFrame->previousFp = getFP();

        /*Initialize the frame to execute the given method*/
        newFrame->thisMethod = thisMethod;
        newFrame->syncObject = NIL; /* Initialized later if necessary*/

        /*Change virtual machine registers to execute the new method*/
        setFP(newFrame);
        setSP((cell*)(newFrame + 1) - 1);
        setIP(thisMethod->u.java.code);
        setCP(thisMethod->ofClass->constPool);


    END ASSERTING NO ALLOCATION
}
```

### 2．popFrame操作

函数 popFrame()用于删除一个执行帧栈及调用当前执行方法时重启执行方法。

```
void popFrame()
{
    /*复位虚拟机寄存器去继续前向方法的执行*/
    POPFRAMEMACRO
}
```

### 19.1.6　JVM 碎片回收堆

垃圾回收是 Java 语言最重要的特点之一。在 Java 语言中，除了 new 语句外没有其他方法为对象申请和释放内存。Java 程序中对内存进行释放和回收的工作是由运行系统 Java 承担的，即 JVM 来完成。在 Sun 公司开发的 Java 解释器和 Hot Java 环境中，采用后台线程来执行碎片回收。这样不仅提高了运行系统的性能，而且使程序设计人员摆脱了动态管理内存的风险。

JVM 有两种类型的存储区：常量缓冲区和方法区。其中常量缓冲区用于存储类名称、方法和字段名称以及串常量，方法区则用于存储 Java 方法的字节码。对于这两种存储区域具体实现方式，在 JVM 规格中没有明确规定。因此 Java 应用程序的存储布局只能在运行过程中确定，完全依赖于具体平台的实现方式。

### 19.1.7　JVM 异常抛出和异常捕获

异常抛出会使当前方法异常结束。将类的异常 Handler 放在类文件的一个表中，当异常发生时，JVM 会从存放异常 Handler 的表中找到合适的异常处理执行，如果当前方法没有合适的处理对应当前异常 Handler，则将当前方法的 Frame 弹出，扔掉 Operand stack 和局部变量，返回到当前方法的调用者中，再重复前面的过程，直到到达调用链表的顶端。如果最外层的方法也没有合适的 Handler，就退出当前线程。

当发生异常时，Java 虚拟机采取进行异常捕获的细节如下：

- 检查与当前方法关联的 catch 子句表，找到与异常匹配的 catch 子句。每个 catch 子句包含其有效指令范围、能够处理的异常类型，以及处理异常的代码块地址。
- 与异常相匹配的 catch 子句应该符合下面的条件：造成异常的指令在其指令范围之内，发生的异常类型是其能处理的异常类型的子类型。如果找到了与该异常匹配的 catch 子句，那么系统转移到异常处理块处进行处理；如果没有找到异常处理块，重复寻找与该异常匹配的 catch 子句的过程，直到检查当前方法的所有嵌套的 catch 子句。
- 由于虚拟机从首个匹配 catch 子句处继续执行，所以 catch 子句表中的顺序是很重要的。Java 代码为结构化的，因此总可以把某个方法的所有的异常处理都按顺序排列在一个表中，对任何可能的程序计数器的值，都能按线性的顺序找到合适的异常处理块，以处理在该程序计数器值下发生的异常情况。
- 如果找不到匹配的 catch 子句，那么当前方法得到一个“未截获异常”的结果并返回到当前方法的调用者，就像异常刚刚在其调用者中发生一样。如果在调用者中仍然没有找到相应的异常处理块，那么这种错误将被传播下去。如果最终错误被传播到最顶层，那么系统最后将调用默认的异常处理块。

KVM 是 JVM 的一种实现，多用于移动电话、PDA 等。KVM 相对其他两种虚拟机最明显的特点是小而高效，比较适合用于嵌入式环境中。接下来将介绍 KVM 实现 JVM 主要功能的细节。

# 19.2　类　装　载

class 文件成为虚拟机上运行的 Java 程序的一部分，要经过三个步骤：装载→连接→初始化。下面将介绍装载类的结构体和相关的主要操作。

## 19.2.1　装载类的结构体

在实现类装载时，KVM 定义了几个数据结构 struct classStruct、struct instanceClassStruct 和 struct arrayClassStruct，这几个数据结构用来保存类的相关信息。

结构体 classStruct 如下：

```
struct classStruct {
    COMMON_OBJECT_INFO(INSTANCE_CLASS)
    UString packageName;                /*最后符号'/'之前的所有内容*/
    UString baseName;                   /*最后符号'/'之后的所有内容*/
    CLASS   next;                       /*哈希表的下一项*/

    unsigned short accessFlags;         /*访问信息*/
    unsigned short key;                 /*类关键字*/
};
typedef struct classStruct*        CLASS;
```

结构体 instanceClassStruct 定义如下：

```
struct instanceClassStruct {
    struct classStruct clazz;           /*公共信息*/

    /* instance classes 的专有信息 */
    INSTANCE_CLASS superClass;          /*超类对象*/
    CONSTANTPOOL constPool;             /*常量池指针*/
    FIELDTABLE  fieldTable;             /*实例变量表指针*/
    METHODTABLE methodTable;            /*虚拟方法表指针*/
    unsigned short* ifaceTable;         /*接口表指针*/
    POINTERLIST staticFields;           /*保持类的静态域*/
    short   instSize;                   /*类的实例大小*/
    short status;                       /*类的就绪状态*/
    THREAD initThread;                  /*类的初始化线程*/
    NativeFuncPtr finalizer;            /*finalizer 指针*/
};
typedef struct instanceClassStruct* INSTANCE_CLASS;
```

结构体 classStruct 和 instanceClassStruc 都是与 class（可能包括类和接口）有关的信息，不同的是 classStruct 所提供的是一些“外围信息”，包括访问信息和可见性等，是一个 class 区分其他 class 的基本信息；instanceClassStruct 所提供的是一些“内容信息”，是一个类本身所定义的内容，比如方法表、字段表等。

结构体 arrayClassStruct 的定义如下：

```
struct arrayClassStruct {
    struct classStruct clazz;                /*公共信息*/
    /*数组类成员专有信息*/
    union {
        CLASS elemClass;        /*数组对象成员类*/
        long primType;          /*原数组的成员类型*/
    } u;
    long itemSize;              /*单个成员的大小*/
    long gcType;                /*垃圾收集的类型，GCT_ARRAY 或 GCT_OBJECTARRAY*/
    long flags;
};
typedef struct arrayClassStruct*    ARRAY_CLASS;
```

## 19.2.2　装载类的操作

上面介绍了类装载保存的结构体，下面介绍装载类的几个操作函数 LoadClass Locally()、LoadClassFromFile()、LoadClassFromZip()，3 个函数分别从不同的方式装载类。

### 1．函数LoadClassFromFile()

函数 LoadClassFromFile()从磁盘装载一个解析器要求的类文件（.class 格式文件），通过创建一个类块结构体装载编译好的类。

```
static ClassClass * LoadClassFromFile(char *fn, char *dir, char *class_name)
{
    extern int OpenCode(char *, char *, char *, struct stat*);
    struct stat st;
    ClassClass *cb = 0;
    int codefd = -1;
    unsigned char *external_class;
    char *detail;
        codefd = OpenCode(fn, NULL, dir, &st);      /*打开一个.class 文件*/
    /*打开失败则退出*/
    if (codefd < 0)                                 /*打开失败*/
        return 0;
    /*将文件载入内存*/
    external_class = (unsigned char *)sysMalloc(st.st_size);
    if (external_class == 0)
        goto failed;
    /*系统 API，读取 codefd 中的内容到 external_class ，大小为 st.st_size ，如果读
    取的大小不等于 st.st_size ，则失败，程序跳转到 failed 处。*/
    if (sysRead(codefd, external_class, st.st_size) != st.st_size)
        goto failed;
       sysClose(codefd);                      /*系统 API 关闭 codefd*/
    codefd = -1;
        cb = allocClassClass();                /*创建内部类*/
    if (cb == NULL ||
        !createInternalClass(external_class, external_class + st.st_size,
               cb, NULL, class name, &detail)) {
        sysFree(external_class);               /*创建失败则释放 external_class*/
        goto failed;
```

```
}
        sysFree(external_class);          /*系统 API 释放 external_class*/
return cb;
/*打开.class 文件出错时，关闭 codefd，创建内部类出错时，释放 cb*/
failed:
    if (codefd >= 0)
        sysClose(codefd);
    if (cb != 0)
        FreeClass(cb);
    return 0;
}
```

### 2．函数LoadClassFromZip()

函数 LoadClassFromZip()通过 zip 格式文件装载类。函数 LoadClassFromFile()与 LoadClassFromZip()基本类似，只是两者读取的文件格式不同而已，前者为.class 格式，后者为.jar 格式。

```
static ClassClass * LoadClassFromZip(zip_t *zipEntry, char *class_name)
{
    ClassClass *cb = 0;
    JAR_DataStreamPtr jdstream = NULL;
    unsigned char *external_class;
    int data_length;
    char *detail;
        jdstream = loadJARfile (zipEntry, class_name); /*导入.jar 格式文件*/
    /*失败则程序跳转到 failed 处*/
    if (jdstream == NULL)
        goto failed;
    /*指定导入的数据和长度*/
    external_class = jdstream->data;
    data_length = jdstream->dataLen;
        cb = allocClassClass();                   /*创建内部类*/
    if (cb == NULL ||
        !createInternalClass(external_class, external_class + data_length,
                cb, NULL, class_name, &detail)) {
        goto failed;
    }
if (jdstream != NULL) {
    /*导入.jar 格式文件失败，则释放 jdstream*/
        freeBytes(jdstream);
    }
return cb;
/*创建内部类失败，则释放 cb；导入.jar 格式文件失败，则释放 jdstream*/
failed:
    if (cb != 0)
        FreeClass(cb);
    if (jdstream != NULL) {
        freeBytes(jdstream);
    }
    return 0;
}
```

### 3．函数LoadClassLocally()

函数 LoadClassLocally()，输入参数为指向类文件的本地路径，其中会调用函数

LoadClassFromFile()。

```
ClassClass *LoadClassLocally(char *name)
{
    ClassClass *cb = 0;
    cpe_t **cpp;
    /*对路径格式进行检验*/
    if (name[0] == DIR_SEPARATOR || name[0] == SIGNATURE_ARRAY)
        return 0;
    /*获取系统环境的 classpath，通过调用系统函数 getenv()*/
    for (cpp = sysGetClassPath(); cpp && *cpp != 0; cpp++) {
        cpe_t *cpe = *cpp;
        char *path;
        /*classpath 元素类型为一个路径或 zip 文件*/
        if (cpe->type == CPE_DIR) {
            path = (char *)sysMalloc(strlen(cpe->u.dir)
                + sizeof(LOCAL_DIR_SEPARATOR)
                + strlen(name)
                + strlen(JAVAOBJEXT)
                + 2);  /*2 is for the . and the \0*/
            if (sprintf(path,
                 "%s%c%s." JAVAOBJEXT, cpe->u.dir,
                 LOCAL_DIR_SEPARATOR, name) == -1) {
                sysFree(path);
                return 0;
            }
            /*调用函数 LoadClassFromFile ()装载类*/
            if ((cb = LoadClassFromFile(sysNativePath(path),
                                  cpe->u.dir, name))) {
                sysFree(path);
                return cb;
            }
        } else if (cpe->type == CPE_ZIP) {
            if (JAR_DEBUG && verbose)
                jio_fprintf(stderr, "Loading classes from a ZIP file... \n");
            /*如果为 zip 文件，则调用函数 LoadClassFromZip ()装载类*/
            if ((cb = LoadClassFromZip(cpe->u.zip, name))) {
                return cb;
            }
        }
    }
    return cb;
}
```

## 19.3　垃圾回收

垃圾回收是 Java 语言的一个特点，这一特点正是 JVM 的设计者通过内存管理来实现的。内存管理分为内存分配和垃圾回收两部分，垃圾回收的技术和策略会严重影响系统的效率，因此垃圾回收是 JVM 设计的重点。下面讲解 KVM 中采用的垃圾回收策略。

### 19.3.1　mark-and-sweep 回收算法

KVM 中的 mark-and-sweep 回收算法分为两个阶段，第一阶段为 mark 阶段，垃圾收集器从 root set 开始搜索，标记每个可达的对象。第二个阶段为 sweep 阶段，垃圾收集器从内存空间的起始地址往后查找，回收那些没有在第一阶段标记的对象所占有的空间，回收的空间加入到内存可用列表中。其实现细节如下：

#### 1．垃圾收集函数garbageCollect()

在函数 garbageCollect()中实现 mark-and-sweep 回收算法。mark-and-sweep 回收算法核心部分为 garbageCollectForReal()，函数 garbageCollect()首先保存虚拟机活动线程调用 garbageCollectForReal()执行垃圾回收，然后恢复环境。

```
void garbageCollect(int moreMemory)
{
    if (gcInProgress != 0) {
        /*不允许循环调用垃圾回收操作*/
        fatalVMError(KVM_MSG_CIRCULAR_GC_INVOCATION);
    }
    gcInProgress++;
    /*等待所有的异步 I/O 完成*/
    RundownAsynchronousFunctions();
    if (ENABLEPROFILING && INCLUDEDEBUGCODE) {
        checkHeap();
    }
    MonitorCache = NULL;            /*清除所有临时监视器*/
    /*在垃圾收集前，保存当前活动的线程的虚拟机寄存器*/
    if (CurrentThread) {
        storeExecutionEnvironment(CurrentThread);
    }
    /*该函数为 mark-and-sweep 回收算法实现核心部分，后面给出其实现的主要代码*/
    garbageCollectForReal(moreMemory);
    /*完成垃圾收集后，载入执行环境*/
    if (CurrentThread) {
        loadExecutionEnvironment(CurrentThread);
    }
    /*允许异步 I/O 继续*/
    RestartAsynchronousFunctions();
    /*恢复垃圾收集结束标志*/
    gcInProgress = 0;
}
```

#### 2．垃圾收集函数garbageCollectForReal()

函数 garbageCollectForReal()是垃圾收集算法 mark-and-sweep 的关键实现部分。函数的标记阶段完成的内容是首先从 root 开始标记对象，然后搜索可达对象，接着标记弱指针列表，将标记为弱引用的对象清除。Sweep 阶段完成释放空间并形成较大的空闲空间。

```
void garbageCollectForReal(int realSize)
{
    CHUNK firstFreeChunk;
```

```
    long maximumFreeSize;
    /*下面是垃圾收集算法实现部分*/
    markRootObjects();              //标志垃圾收集的 root 对象
    markNonRootObjects();           //查找堆栈搜索仅从其他堆栈对象可达的对象
    markWeakPointerLists();         //标志弱指针列表
markWeakReferences();               //标记弱引用对象，弱引用对象将被清除
/*实现 mark-and-sweep 算法的 sweep 阶段，释放没有活动的方法所占用的堆栈空间*/
firstFreeChunk = sweepTheHeap(&maximumFreeSize);
/*compact 阶段，通过内存的移动构建较大的可用空间*/
#if ENABLE HEAP COMPACTION
    if (realSize > maximumFreeSize) {
        /*对堆栈实行紧缩操作，获得可用空间*/
        breakTableStruct currentTable;
        cell* freeStart = compactTheHeap(&currentTable, firstFreeChunk);
        /*得到可用空间后，进行更新操作，更新 root 对象表和堆栈对象表*/
        if (currentTable.length > 0) {
            updateRootObjects(&currentTable);
            updateHeapObjects(&currentTable, freeStart);
        }
        if (freeStart < CurrentHeapEnd - 1) {
            firstFreeChunk = (CHUNK)freeStart;
            firstFreeChunk->size =
                (CurrentHeapEnd - freeStart - HEADERSIZE) << TYPEBITS;
            firstFreeChunk->next = NULL;
        } else {
            /*内存完全满时，没有多余的空间可以通过移动内存来获得*/
            firstFreeChunk = NULL;
        }
    }
#endif
    FirstFreeChunk = firstFreeChunk;
}
```

KVM 中采用了 mark-and-sweep 回收算法，还存在其他分代回收算法和增量收集算法，下面给出其简单介绍。

## 19.3.2　分代回收算法

分代回收算法是根据对象存在时间长短将对象进行分类，每个子堆为一代。随着对象的消亡，垃圾回收器从最年轻的子堆开始回收对象。

分代回收算法在一定程度上降低了垃圾回收给应用带来的负担，使应用的吞吐量达到极限值。但是无法解决垃圾收集所带来的应用暂停。在一些对实时性要求很高的应用场景下，垃圾收集暂停所带来的请求堆积和请求失败是无法接受的。如话音业务，可能要求请求的返回时间在几百甚至几十毫秒内，如果分代垃圾回收算法要达到该指标，只能把最大堆的设置限制在一个相对较小范围内，但是这样又限制了应用本身的处理能力，同样也是不可接收的。

分代垃圾回收算法为考虑实时性要求而提供了并发回收器，支持最大暂停时间的设置，但是受到分代垃圾回收的内存划分模型限制，其效果并不理想。

为了达到实时性的要求（其实 Java 语言最初的设计也是在嵌入式系统上的），一种新垃圾回收方式要求既支持短的暂停时间，又支持大的内存空间分配。这样可以很好地解决

分代方式带来的问题，增量收集可以满足上述要求。

### 19.3.3　增量收集

增量收集的方式在理论上可以解决分代收集方式带来的问题。增量收集把堆空间划分成一系列内存块，使用时，先使用其中一部分，垃圾收集时把之前使用部分中的存活对象再放到后面没有使用的空间中，因此可以实现边使用边收集的效果，避免了分代方式使用完整个内存空间再暂停回收的情况。

分代收集方式中也支持并发收集，分代收集的缺点就是把整个堆作为一个内存块，这样一方面会造成碎片（无法压缩），另一方面它的每次收集都是对整个堆的收集，无法进行选择，在暂停时间的控制上很弱。而增量收集方式，通过内存空间的分块，恰恰可以解决上述问题。

## 19.4　解　析　器

KVM 的解析器就是将字节码解析成具体平台上的操作指令执行。解析器由循环执行函数 Interpret()，函数 Interpret()中真正执行解析功能的函数为 FastInterpret()，下面对这两个函数进行分析。

### 19.4.1　函数 Interpret()

函数 Interpret()在函数 KVM_Start()中被调用。在该函数的 catch 尾部调用 END_CATCH_AND_GOTO(startTry)，该函数相当于执行 goto 功能跳转到 startTry 处，因此在该函数体内循环执行函数 FastInterpret()进行解析工作。

```
void Interpret() {
    /*解析器的入口*/
startTry:
    CurrentNativeMethod = NULL;
    TRY {
        START_TEMPORARY_ROOTS
            IS_TEMPORARY_ROOT(thisObjectGCSafe, NULL);
            FastInterpret();
        END_TEMPORARY_ROOTS
    } CATCH (e) {
        START_TEMPORARY_ROOTS
            IS_TEMPORARY_ROOT(e, e);
            throwException(&e);
        END_TEMPORARY_ROOTS
/*END_CATCH_AND_GOTO(startTry)相当于执行 goto startTry 操作，程序又跳转到
startTry 处，相当于循环函数*/
} END_CATCH_AND_GOTO(startTry)
/*打印解析信息*/
#if INSTRUMENT
```

```
    fprintf(stdout,"bytecodes     =\t%ld\n", bytecodes);
    fprintf(stdout,"slowcodes  =\t%ld\t(%ld)%%\n", slowcodes,slowcodes/
    (bytecodes/100));
    fprintf(stdout,"calls  =\t%ld\t(%ld)%%\n", calls, calls/(bytecodes/
    100));
    fprintf(stdout,"branches taken =\t%ld\t(%ld)%%\n", branches,branches/
    (bytecodes/100));
    fprintf(stdout,"rescheduled=\t%ld\t(%ld)%%\n", reshed,reshed/
    (bytecodes/100));
#endif /* INSTRUMENT */
}
```

### 19.4.2　函数 FastInterpret()

函数 FastInterpret()是执行解析的真正实体函数。下面为函数 FastInterpret()的主要部分，留下函数的主干部分，便于清楚函数的整体实现。函数 FastInterpret()的关键部分为 switch 结构部分，该部分根据寄存器变量 token 的值，执行 callMethod_general 部分的代码，对 Java 程序中的函数进行解析，包括将函数解析为具体平台的本地函数。

```
void FastInterpret() {
   /*解析器需要定义的局部变量*/
    METHOD   thisMethod;
    OBJECT   thisObject;
    int      invokerSize;
    const char *exception;
#if ENABLE JAVA DEBUGGER
    register BYTE token;                    /*寄存器变量*/
#endif
/* IP 程序计数器*/
next3:  ip++;
next2:  ip++;
next1:  ip++;
next0:
#if ENABLE_JAVA_DEBUGGER
    token =*/ip;                            /*用于保存程序计数器 IP 内容*/
    __doSingleStep()
next0a:
#endif
#endif /*RESCHEDULEATBRANCH*/
   /*调用函数 InstructionProfile()*/
    INSTRUCTIONPROFILE
   /*打开指令跟踪*/
    INSTRUCTIONTRACE
   /*增量式字节码计数器*/
    INC BYTECODES
   /*在每个字节码之前垃圾回收选项*/
    DO VERY EXCESSIVE GARBAGE COLLECTION
   /*指派字节码，通过 switch 分支方式实现解析*/
#if ENABLE JAVA DEBUGGER
    switch (token) {
#else
    switch (((unsigned char)*ip)) {
#endif
        callMethod_interface:
```

```
        invokerSize = 5;                    /*设置字节码的大小为 5*/
        goto callMethod general;
    callMethod_virtual:
    callMethod_static:
    callMethod special:
        invokerSize = 3;                    /*设置字节码的大小为 3*/
    /* callMethod_general 部分的代码即对 Java 程序中的函数进行解析，包括将函数
    解析为具体平台的本地函数*/
    callMethod_general: {
        INC CALLS
        /*检查方法是否为本地方法，实际上 thisMethod 也就是编译 Java 程序中的某个函
        数后得到的方法*/
        if (thisMethod->accessFlags & ACC_NATIVE) {
            ip += invokerSize;
            VMSAVE
            invokeNativeFunction(thisMethod);
            VMRESTORE
            TRACE_METHOD_EXIT(thisMethod);
            goto reschedulePoint;
        }
        /*检查方法是否为抽象方法*/
        if (thisMethod->accessFlags & ACC ABSTRACT) {
            VMSAVE
            raiseExceptionWithMessage(AbstractMethodError, methodName
            (thisMethod));
            VMRESTORE
        }
        thisObjectGCSafe = thisObject;
        VMSAVE
        pushFrame(thisMethod);                  //方法入栈
        VMRESTORE
        /*执行到下一条指令返回*/
        fp->previousIp += invokerSize;
        /*检查该方法是否为同步方法*/
        if (thisMethod->accessFlags & ACC_SYNCHRONIZED) {
            VMSAVE
            monitorEnter(thisObjectGCSafe);
            VMRESTORE
            fp->syncObject = thisObjectGCSafe;
        }
        thisObjectGCSafe = NULL;
        goto reschedulePoint;
    }
    /*接下来执行各种异常的处理，空指针异常、数组越界异常、算术异常等*/
    handleNullPointerException: {
        exception = NullPointerException;
        goto handleException;
    }
    handleArrayIndexOutOfBoundsException: {
        exception = ArrayIndexOutOfBoundsException;
        goto handleException;
    }
    handleArithmeticException: {
        exception = ArithmeticException;
        goto handleException;
    }
    handleArrayStoreException: {
        exception = ArrayStoreException;
        goto handleException;
```

```
        }
        handleClassCastException: {
            exception = ClassCastException;
            goto handleException;
        }
        handleException: {
            VMSAVE
            raiseException(exception);
            VMRESTORE
            goto reschedulePoint;
        }
        default: {
            sprintf(str_buffer, KVM_MSG_ILLEGAL_BYTECODE_1LONGPARAM,
                    (long)TOKEN);
            fatalError(str buffer);
            break;
        }
    }
    fatalError(KVM_MSG_INTERPRETER_STOPPED);
}
```

### 19.4.3　函数 SlowInterpret ()

函数 SlowInterpret ()在函数 FastInterpret()中被调用，SlowInterpret()作为次级解析器。其完成的功能在 FastInterpret()函数中基本可以完成，SlowInterpret()相当于 FastInterpret()子函数，两者的风格也类似。

```
void SlowInterpret(ByteCode token) {
    METHOD   thisMethod;
    OBJECT   thisObject;
    int      invokerSize;
    switch (token) {
        callMethod_interface: {
            invokerSize = 5;                /*设置字节码的大小为 5*/
            goto callMethod general;
        }
        callMethod_virtual:
        callMethod_static:
        callMethod special:
            invokerSize = 3;                /*设置字节码大小为 3*/
        callMethod general: {
            /*callMethod_general 部分代码即对 Java 程序中的函数进行解析，包括将函
            数解析为具体平台的本地函数*/
        }
       /*执行各种异常处理*/
       handleXXXXException(){
       }
next3:  ip++;
next2:  ip++;
next1:  ip++;
next0:
reschedulePoint:
    return;
}
```

# 19.5　Java 编程浅析

本节的目标只是让读者了解 Java 编程中的基本规则和概念，包括如何编译源文件如何命名、类如何定义、主函数如何定义，以及如何编译 Java 程序、如何执行解析等。讲解本节是为分析 19.6 节的 KVM 运行做铺垫，如果读者熟悉 Java 编程可以跳过本节。

## 19.5.1　Java 程序命令

Java 程序源文件采用.class 后缀命名，源文件的名字与类的名字保持一致。在编程风格上 Java 类的首字母采用大写，方法和属性首字母采用小写方式，命令都采用驼峰式方式。下面举例说明其编程方式：

```
public class TestKVM {                                  //类的定义
    private String theStates;                           //属性的定义
    public String getTheStates() {                      //方法的定义
        return theStates;
    }
    public void setTheStates(String theStates) {
        this.theStates = theStates;
    }
}
```

在 Java 程序中访问的权限 private、public 都放在每个方法和属性前进行单独指定，与 C++有区别。

## 19.5.2　Java 构造函数

下面为构造函数的编写方式，如果是继承类有时在构造函数中显示调用父类时采用 super()，示例如下：

```
    public TestKVM(String theStates) {      //构造函数定义，参数为成员变量
        super();
        this.theStates = theStates;
    }
```

构造函数中可以根据需要自定成员变量作为参数，也可以通过参数不同定义多个构造函数。

## 19.5.3　Java 主函数

Java 的编程思想是：一切皆属于类。完全是面向对象的思想，比 C++更彻底。所以其主函数 main()也是放在类中的。下面给出 main()函数的定义方式：

```
public class TestKVM {
    String theStates;
    public String getTheStates() {
        return theStates;
    }
```

```
    public void setTheStates(String theStates) {
        this.theStates = theStates;
    }
    public TestKVM(String theStates) {
        this.theStates = theStates;
    }
    /**
     * 注释的编写方式                                   //注释的编写方式
     */
    public static void main(String[] args) {      //主函数的编写方式
        TestKVM t=new TestKVM("kvm");             //对象实例化
        System.out.println(t.getTheStates());     //打印输出
    }
}
```

编写完后命名为 TestKVM.java。

### 19.5.4　Java 程序编译与运行

编译 Java 程序采用 Java 编译器 javac，运行 Java 程序也就是通过解析器执行解析 Java 程序，将 Java 字节码解析成具体平台的字节码运行，其解析器为 Java。编译和运行的方式如下：

```
# javac TestKVM.java
# java TestKVM
```

执行结果为：kvm

## 19.6　KVM 执行过程

前面从 Java 虚拟机的各个部分分析其主要功能，也了解了 Java 编译和 Java 运行的方法。在前面这些知识的铺垫下，本节将从整体上分析虚拟机的运行过程，通过 KVM 虚拟机展示一般虚拟机执行的整个过程。下面分别介绍 KVM 执行过程中的主要步骤：

### 19.6.1　KVM 启动过程

在了解 KVM 执行过程前，读者可以先看 KVM 的运行状态图，在接下来的分析过程中主要围绕该状态图进行。该状态图如图 19.2 所示。

KVM 执行过程在函数 KVM_Start()中体现，函数 KVM_Start()的代码如下：

```
int KVM_Start(int argc, char* argv[])
{
    ARRAY arguments;
    INSTANCE_CLASS mainClass = NULL;
    volatile int returnValue = 0; /* Needed to make compiler happy */
    /*通过 try…catch 机制捕获异常*/
    TRY {
        VM_START {
            /*创建 ROM 映像，KVM 会先把 class 文件转化为 C 语言源代码，然后编入 KVM 可
             执行程序中，这样当使用系统类库时，KVM 就不再访问外部的 class 文件*/
```

```
            CreateROMImage();
            /*初始化浮点数算术*/
            InitializeFloatingPoint();
#if ASYNCHRONOUS_NATIVE_FUNCTIONS
            /*初始化异步 I/O 系统*/
            InitalizeAsynchronousIO();
#endif

            /*初始化虚拟机运行时的基本结构*/
            InitializeNativeCode();
            InitializeVM();
            /*初始化全局变量*/
            InitializeGlobals();
            /*初始化 profiling 变量*/
```

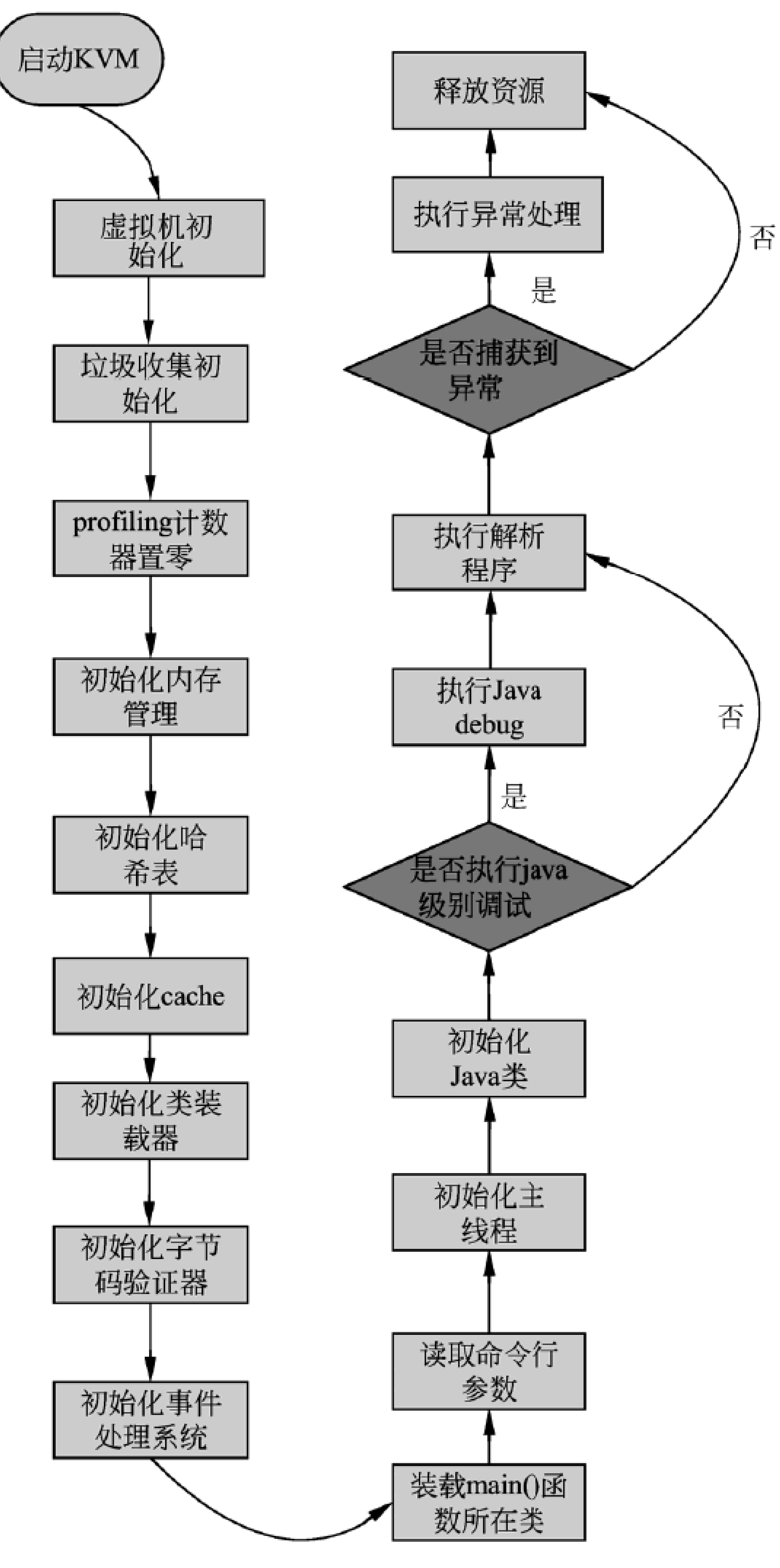



图 19.2　KVM 执行状态图

```
            InitializeProfiling();
            /*初始化内存管理*/
            InitializeMemoryManagement();
            /*初始化内部哈希表*/
            InitializeHashtables();
            /*创建主缓冲区*/
            InitializeInlineCaching();
            /*创建动态类装载器*/
            InitializeClassLoading();
            /*导入并初始化虚拟机需要的类*/
            InitializeJavaSystemClasses();
            /*初始化类文件验证器*/
            InitializeVerifier();
            /*初始化事件处理系统*/
            InitializeEvents();
            /*导入主函数所在的类*/
            mainClass = loadMainClass(argv[0]);
            /*解析命令行参数*/
            arguments = readCommandLineArguments(argc - 1, argv + 1);
            /*通过命令行参数和主类初始化主线程*/
            InitializeThreading(mainClass, arguments);
#if ENABLE_JAVA_DEBUGGER
            /*为 Java 代码级别的调试准备好虚拟机*/
            if(debuggerActive) {
                InitDebugger();
            }
#endif
            /*在没有明确执行新指令时创建以下几个 Java 类，并按如下顺序入栈，即栈底部为
            类 JavaLangOutOfMemoryError，栈顶为 JavaLangClass*/
            initializeClass(JavaLangOutOfMemoryError);
            initializeClass(JavaLangSystem);
            initializeClass(JavaLangString);
            initializeClass(JavaLangThread);
            initializeClass(JavaLangClass);
#if ENABLE_JAVA_DEBUGGER
            /*准备 Java 代码级别的调试*/
            if(vmDebugReady) {
                setEvent VMInit();
                if(CurrentThread == NIL) {
                    CurrentThread = removeFirstRunnableThread();
                    /*导入执行环境，即通过将线程的执行环境保存在寄存器中*/
                    loadExecutionEnvironment(CurrentThread);
                }
                sendAllClassPrepares();
            }
#endif /*ENABLE_JAVA_DEBUGGER*/
            /*开始执行解析*/
            Interpret();
        } VM_FINISH(value) {
            returnValue = value;
        } VM_END_FINISH
    } CATCH(e) {
        /*捕获上面没有捕获的异常*/
        if(mainClass == NULL) {
            /*如果主函数不存在，则打印专门的消息*/
            char buffer[STRINGBUFFERSIZE];
            sprintf(buffer, "%s", getClassName((CLASS)e->ofClass));
```

```
            if (e->message != NULL) {
                sprintf(buffer + strlen(buffer),": %s", getStringContents
                (e->message));
            }
            sprintf(str buffer, "%s", buffer);
            AlertUser(str_buffer);
            returnValue = 1;
        } else {
            /*无法捕捉的异常记录在日志中*/
            Log->uncaughtException(e);
            returnValue = UNCAUGHT EXCEPTION EXIT CODE;
        }
    } END CATCH
    return returnValue;
}
```

## 19.6.2　KVM 用到的计数器清零

函数 InitializeProfiling()用于将 KVM 用到的计数器清零。该函数中列出了 KVM 使用的计数器，代码如下：

```
void InitializeProfiling()
{
    /*所有计数器清零*/
    InstructionCounter          = 0;            /*指令计数器*/
    ThreadSwitchCounter         = 0;            /*线程 Switch 计数器*/
    DynamicObjectCounter        = 0;            /*动态对象计数器*/
    DynamicAllocationCounter    = 0;            /*动态内存分配计数器*/
    DynamicDeallocationCounter = 0;             /*动态内存重分配计数器*/
    GarbageCollectionCounter    = 0;            /*垃圾收集计数器*/
    TotalGCDeferrals            = 0;            /*延迟回收的垃圾收集对象总数*/
    MaximumGCDeferrals          = 0;            /*延迟回收的垃圾收集对象最大数目*/
    GarbageCollectionRescans    = 0;
#if ENABLEFASTBYTECODES
/*Cache 的使用会大大改善系统性能，其原理为程序的局部性原理（即程序的地址访问流有很强的
时序相关性，未来的访问模式与最近已发生的访问模式相似）。根据这一局部性原理，把主存储器
中访问概率最高的内容存放在 Cache 中，当 CPU 需要读取数据时就首先在 Cache 中查找是否有所
需内容，如果有则直接从 Cache 中读取；若没有再从主存中读取该数据，然后同时送往 CPU 和
Cache*/
    InlineCacheHitCounter       = 0;            /*访问高速缓冲区的次数*/
    InlineCacheMissCounter      = 0;            /*访问高速缓冲区失败的次数*/
    MaxStackCounter             = 0;            /*需要栈空间的最大数目*/
#endif
#if USESTATIC
    StaticObjectCounter         = 0;                /*静态对象数目*/
    StaticAllocationCounter     = 0;                /*静态空间分配的大小*/
#endif
}
```

### 19.6.3　KVM 初始化内存管理

在 KVM 中执行前期工作主要用于初始化一些相关的资源，初始化过程包括虚拟的硬件和软件的初始化过程。在写 Java 程序时，不需要程序员释放空间，虚拟机会自动帮程序员进行垃圾回收，下面分析虚拟机 KVM 如何初始化内存管理部分。

（1）函数 InitializeMemoryManagement()用于初始化内存管理。初始化内存时，先初始化堆，然后为第一个对象分配空间。

```
void InitializeMemoryManagement(void)
{
    int index;
gcInProgress = 0;
/*初始化堆*/
    InitializeHeap();
    index = 0;
    GlobalRoots[index++].cellpp = (cell **)&AllThreads;
    GlobalRoots[index++].cellpp = (cell **)&CleanupRoots;
    GlobalRootsLength = index;
    TemporaryRootsLength = 0;
    /*为第一个对象分配空间*/
    CleanupRoots =

(POINTERLIST)callocObject(SIZEOF_POINTERLIST(CLEANUP_ROOT_SIZE),
                                    GCT_POINTERLIST);
}
```

（2）函数 InitializeHeap()用于初始化堆栈，在该函数中调用函数 allocateHeap()分配堆栈大小 RequestedHeapSize。该堆栈可以根据需要动态分配空间，而不是一个固定的空间。

```
void InitializeHeap(void)
{
    /*堆栈大小*/
    VMHeapSize = RequestedHeapSize;
    /*指向堆栈的底部 cell 指针*/
    AllHeapStart = allocateHeap(&VMHeapSize, &TheHeap);
    /*分配失败*/
    if (TheHeap == NIL) {
        fatalVMError(KVM_MSG_NOT_ENOUGH_MEMORY);
    }
    /*堆栈通过指向栈底指针、栈顶指针和堆栈大小表示，底部指针偏移堆栈大小的位置就到了堆
    栈的顶部。该堆栈可以根据需要动态增加大小*/
    CurrentHeap    = AllHeapStart;
    CurrentHeapEnd = PTR_OFFSET(AllHeapStart, VMHeapSize);
/*分配 Java 堆空间时，不采用块*/
#if !CHUNKY_HEAP
    FirstFreeChunk = (CHUNK)CurrentHeap;
    FirstFreeChunk->size =
           (CurrentHeapEnd -CurrentHeap - HEADERSIZE) << TYPEBITS;
    FirstFreeChunk->next = NULL;
#endif
    /*如果当前分配的空间为 0，指向当前堆栈顶部指针等于指向所有堆栈的顶部指针*/
#if ENABLE_HEAP_COMPACTION
    PermanentSpaceFreePtr = CurrentHeapEnd;
```

```
#endif
    AllHeapEnd          = CurrentHeapEnd;
/*将分配堆栈情况写入日志*/
#if INCLUDEDEBUGCODE
    if (tracememoryallocation) {
        Log->allocateHeap(VMHeapSize, (long)AllHeapStart, (long)
        AllHeapEnd);
    }
    NoAllocation = 0;
#endif
}
```

（3）再分配堆栈函数 allocateHeap()，参数 sizeptr 为分配堆栈空间的大小，参数 realresultptr 返回成功分配的堆栈指针。

```
cell *allocateHeap(long *sizeptr, void **realresultptr) {
    /*分配堆栈大小，返回分配的空间指针为 space*/
    void *space = malloc(*sizeptr + sizeof(cell) - 1);
    *realresultptr = space;
    return (void *) ((((long)space) + (sizeof(cell) - 1)) & ~(sizeof(cell)
    - 1));
}
```

## 19.6.4　KVM 中的哈希表初始化

KVM 中创建了 3 个哈希表，表 UTFStringTable 用于存储所有的 utf C 字符串，表 InternStringTable 存储所有用到 Java 字符串，表 ClassTable 用于存储 Java 类。

```
void InitializeHashtables() {
if (!ROMIZING) {
    /*创建用于存储 utf C 字符串的哈希表*/
        createHashTable(&UTFStringTable, UTF_TABLE_SIZE);
    /*创建用于存储 Java 字符串的哈希表*/
        createHashTable(&InternStringTable, INTERN_TABLE_SIZE);
    /*创建用于存储类的哈希表*/
        createHashTable(&ClassTable, CLASS_TABLE_SIZE);
    }
}
```

KVM 对哈希表的定义：

```
typedef struct HashTable {
    long bucketCount;           /*结点个数*/
    long count;                 /*表中所有元素个数*/
    cell *bucket[1];            /*入口指针数组*/
} *HASHTABLE;
/*存储 Java 字符串的哈希表*/
extern HASHTABLE InternStringTable;
/*存储所有的 utf C 字符串的哈希表*/
extern HASHTABLE UTFStringTable;
/*存储 Java 类*/
extern HASHTABLE ClassTable;
```

创建哈希表函数 createHashTable()，参数 tablePtr 用于返回创建哈希表的地址，参数 bucketCount 表示哈希表的结点个数，用于计算哈希表的大小。

```
void createHashTable(HASHTABLE *tablePtr, int bucketCount) {
    /*通过结点个数计算哈希表的大小*/
    int objectSize = SIZEOF_HASHTABLE(bucketCount);
    /*创建哈希表*/
    HASHTABLE table = (HASHTABLE)callocPermanentObject(objectSize);
    /*设置创建的哈希表结点个数*/
    table->bucketCount = bucketCount;
    /*创建的哈希表赋给参数 tablePtr 返回*/
    *tablePtr = table;
}
```

### 19.6.5 KVM 中的事件初始化

函数 InitializeEvents()用于初始化虚拟机的事件系统，方便虚拟机的关闭和重新启动。

```
void InitializeEvents()
{
    waitingThread = 0;          /*等待线程*/
    makeGlobalRoot((cell**)&waitingThread);
                                /*将等待线程存储在全局数组 GlobalRoots 中*/
    eventInP = 0;
    eventCount = 0;
}
```

整个初始化过程完成后，就载入主类，并根据主类和输入命令行参数初始化主线程。接下来创建几个 Java 类并入栈，然后开始执行解析。

### 19.6.6 KVM 中的资源释放

KVM 的资源释放过程与其资源分配过程相对应，执行的顺序与分配的顺序相反。函数 KVM_Cleanup()用于 KVM 的资源释放。

```
void KVM_Cleanup()
{
    FinalizeVM();                   /*结束虚拟机*/
    FinalizeInlineCaching();        /*释放 cache*/
    FinalizeNativeCode();           /*与 InitializeNativeCode()对应*/
    FinalizeJavaSystemClasses();    /*释放 Java 系统类*/
    FinalizeClassLoading();         /*释放载入类*/
    FinalizeMemoryManagement();     /*结束内存管理*/
    DestroyROMImage();              /*销毁 ROM*/
    FinalizeHashtables();           /*删除哈希表*/
}
```

## 19.7 PC 机安装 JVM

前面对 JVM 的主要功能进行了介绍，同时也分析了其类装载器和解析器的实现。那

么 Java 代码通过在一个平台进行编译就应该能在安装了 JVM 的其他平台上运行。下面将介绍 JVM 在 Windows 和 Linux 上的安装过程。

## 19.7.1　JVM 在 Windows 上的安装

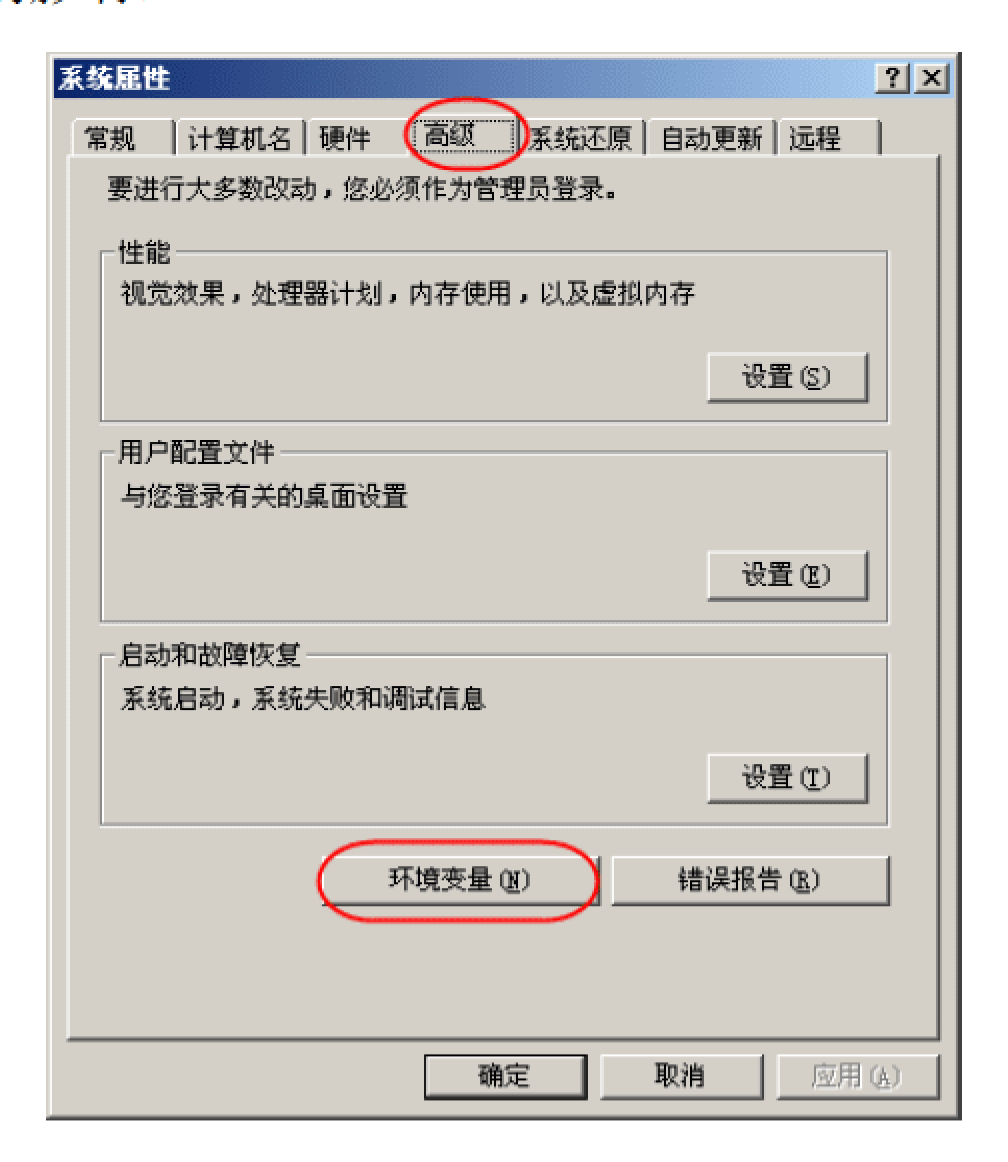
在 Windows 和 Linux 上安装 JVM 都比较简单，这里给出其安装的步骤。

（1）下载 JDK 的 Windows 安装文件，安装在指定的目录下；

（2）设置环境变量。

- 新增的环境变量 JAVA_HOME 其值为 JDK 的安装路径，如 C:\jdk1.6；修改环境变量 path，在 path 值中添加安装 JDK 的 bin 文件路径，如 C:\jdk1.6\bin，这样是让系统能够找到 javac 和 java 等工具。
- 新增环境变量 CLASSPATH 其值设置为 D:\java;.;C:\jdk1.6\lib\tools.jar，该环境变量指定的路径下的 class 文件能被 JVM 识别并载入，前面分析类装载器时已经提到载入.class 文件，这里就是对类装载器中说的 classpath 环境变量进行设置。

Windows 设置环境变量的方法为，右击“我的电脑”，在弹出的快捷菜单中选择“属性”选项，打开“系统属性”对话框。选择“高级”选项卡，在其中单击“环境变量”按钮，如图 19.3 所示。添加新的环境变量可以在环境变量对话框中单击“新建”按钮进行设置，修改环境变量可以先选好要修改的环境变量，然后再单击“编辑”进行修改。一般进行修改时，新版本的安装路径从前面开始添加，如图 19.4 所示，这样是为了防止受旧版本的影响。





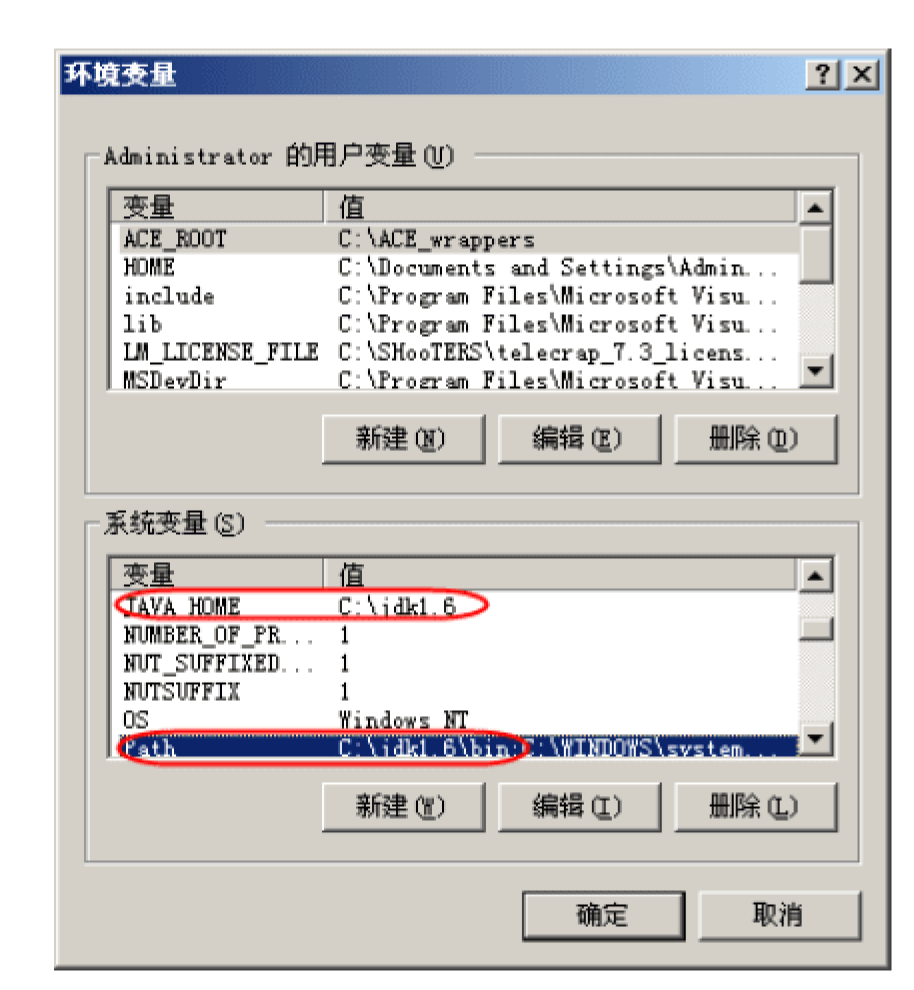




图 19.3　环境变量的设置　　　　图 19.4　环境变量的增加和修改

（3）安装完成并正确设置好环境变量后，可以对安装的结果进行简单测试，打开一个控制台，在其中输入 javac 和 java 命令进行测试。安装正确后，测试结果如图 19.5 所示。

```
命令提示符
C:\Documents and Settings\Administrator>java -version
java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b05)
Java HotSpot(TM) Client VM (build 1.6.0_03-b05, mixed mode, sharing)

C:\Documents and Settings\Administrator>javac -version
javac 1.6.0_03
```

图 19.5　测试 JVM 安装结果

## 19.7.2　JVM 在 Linux 上的安装

JVM 在 Linux 上的安装与 Windows 上的安装方法类似。其安装过程如下：

（1）下载并安装 Linux 版本的 JDK 文件 jdk-6u19-linux-i586.bin，安装方法如下：

```
# ./jdk-6u19-linux-i586.bin
# y                           //接受 license
```

（2）设置环境变量，设置的环境变量也包括 JAVA_HOME、CLASSPATH 和 CLASSPATH，设置的方法如下：

```
# cd $HOME
# vi .bash_profile
//文件.bash_profile 中添加下面的内容
JAVA_HOME=/usr/jdk1.6.0_19
PATH=$JAVA_HOME/bin:$PATH
CLASSPATH=$JAVA_HOME/lib:/usr/JAVA:./
export JAVA_HOME PATH CLASSPATH
```

- /usr/jdk1.6.0_19 为 JDK 的安装目录。
- PATH：添加指令的搜索路径，再使用 javac 编译一个.java 文件，添加该路径后可以直接在命令行中用 javac 而不要采用绝对路径。
- CLASSPATH：为 Java 代码运行的类路径，/usr/JAVA 为 Java 代码生成的 class 所放置的路径。

（3）安装和设置完成后同样可以使用 java –version 和 javac –version 命令进行测试安装的结果，测试结果如图 19.6 所示。

```
root@bogon:~
文件(F)  编辑(E)  查看(V)  终端(T)  标签(B)  帮助(H)
[root@bogon ~]# java -version
java version "1.6.0_19"
Java(TM) SE Runtime Environment (build 1.6.0_19-b04)
Java HotSpot(TM) Client VM (build 16.2-b04, mixed mode, sharing)
[root@bogon ~]# javac -version
javac 1.6.0_19
[root@bogon ~]#
```

图 19.6　测试 Linux 下 JVM 安装结果

# 19.8　KVM 移植和测试

在对 JVM 进行交叉编译时，为了与 KVM 版本一致，Java SDK 的版本选择为 j2sdk-1_4_2。交叉编译器选择低版本的编译器 arm-linux-gcc。

## 19.8.1　SDK 安装和环境变量设置

下载 j2sdk-1_4_2_18-linux-i586.bin 进行安装，j2dk-1_4_2 的其他几个版本也试过，都可以成功编译，读者可以选择任意一个进行试验。安装过程如下：

```
# ./ j2sdk-1_4_2_18-linux-i586.bin
# y                              //同意 license 后进行自动安装
```

安装完成后对环境变量进行修改，修改过程如下：

```
# vi /root/.bash profile
JAVA_HOME=/usr/j2sdk1.4.2_18
PATH=$JAVA HOME/bin:$PATH
CLASSPATH=$JAVA_HOME/lib:/usr/JAVA
export JAVA_HOME PATH CLASSPATH
```

修改完成后，需要重新启动计算机使环境变量生效。

## 19.8.2　修改 Makefile 和代码

对 KVM 需要进行交叉编译，因对对应的 Makefile 中应该修改 gcc 为 arm-linux-gcc。下载的源码为 j2me_cldc-1_1-fcs-src-winunix.zip，在 Linux 下可以直接在 XWindows 模式下通过右击解压，然后复制到/usr 目录中。

### 1．修改kvm的Makefile

进入/usr/j2me_cldc/kvm/VmUnix/build 目录下修改 Makefile，修改过程如下：

```
# cd /usr/j2me cldc/kvm/VmUnix/build
# vi Makefile
```

增加平台定义，可以放在所有条件之前，或者放在 Makefile 的开头。

```
export PLATFORM=linux
```

修改 gcc 为 arm-linux-gcc。

```
ifeq ($(GCC), true)
   CC = gcc
   CFLAGS =  -Wall $(CPPFLAGS) $(ROMFLAGS) $(OTHER_FLAGS)
   DEBUG_FLAG = -g
   OPTIMIZE_FLAG = -O2
```

修改为：

```
ifeq ($(GCC), true)
   CC = arm-linux-gcc
```

```
   CFLAGS =  -Wall $(CPPFLAGS) $(ROMFLAGS) $(OTHER_FLAGS)
   DEBUG_FLAG = -g
   OPTIMIZE_FLAG = -O2
```

### 2．注释掉浮点数功能

修改目录/usr/j2me_cldc/kvm/VmUnix/src 下的文件 runtime_md.c，将浮点数功能注释掉。注释方法如下：

```
/*=========================================================================
===
 * FUNCTION:     InitializeFloatingPoint
 * TYPE:         initialization
 * OVERVIEW:     Called at startup to setup the FPU.
 *
 * INTERFACE:
 *   parameters:  none
 *   returns:     none

*=========================================================================
*/
void InitializeFloatingPoint() {
#if defined(LINUX) && PROCESSOR_ARCHITECTURE_X86
    /* Set the precision FPU to double precision */
//    fpu_control_t cw = (_FPU_DEFAULT & ~_FPU_EXTENDED) | _FPU_DOUBLE; //
可以在此将函数体注释掉
//     _FPU_SETCW(cw);
#endif
}
```

## 19.8.3　KVM 编译

在编译 KVM 前，首先进入 tools/preverifier/build/linux 目录进行编译 preverify，在编译 preverify 时不需要修改该目录下的 Makefile，直接进行编译。

```
#make
```

编译生成 preverify 后回到/usr/j2me_cldc/build/linux 执行 make 编译 KVM，而并非进入目录/usr/j2me_cldc/kvm/VmUnix/build 下进行编译。

```
# make
```

编译完成后，在/usr/j2me_cldc/kvm/VmUnix/build 目录下生成 KVM。

## 19.8.4　测试

在移植前先在 PC 上写个小程序进行测试，确保该程序能在 PC 上顺利通过，对环境变量进行重新修改，主要目的是保证执行命令方便输入，新修改后的环境变量设置如下：

```
JAVA_HOME=/usr/j2sdk1.4.2_18
CLASSPATH=$JAVA_HOME/lib:/usr/JAVA
CLDC_HOME=/usr/j2me_cldc
CLDC_PATH=$CLDC_HOME/bin
```

```
PATH=$JAVA_HOME/bin:$CLDC_PATH/linux:$PATH
export JAVA_HOME PATH CLASSPATH CLDC_HOME CLDC_PATH
```

修改完环境变量后，重新启动计算机或者使用命令 source /root/.bash_profile，使新设置的环境变量生效。

```
# source /root/.bash_profile
```

在进行测试前，首先写个简单的 Java 测试程序，程序很简单，代码如下：

```
public class TestKVM                              //Java 类定义
{
        public static void main(String[] args)  //主函数
        {
                System.out.println("Hello KVM!");//打印消息
        }
}
```

该代码文件命名为 TestKVM 与类同名，保存在/usr/JAVA 目录下。使用 javac 命令进行编译，编译生成的.class 文件放在目录 tmpclass 中。

```
# mkdir tmpclasses
# javac -bootclasspath /usr/j2me cldc/bin/common/api/classes -d tmpclasses
TestKVM.java
```

javac 命令的各项参数说明：

- -bootclasspath 后面跟 KVM/CLDC java 库的类文件位置。
- /usr/j2me_cldc/bin/common/api/classes KVM/CLDC java 库的类文件路径。
- -d 后面跟输出文件路径。
- tmpclasses 输出编译好的.class 文件路径。
- TestKVM.java 待编译的 Java 程序文件。

编译完成后检查生成的.class 文件。

```
# ls tmpclasses
```

在 tmpclasses 目录下生成文件 TestKVM.class，接下来执行预验证操作。

```
# preverify -classpath /usr/j2me_cldc/bin/common/api/classes:tmpclasses
-d . TestKVM
```

preverify 命令的各项参数说明如下：

- -classpath 后面跟 KVM/CLDC java 库的类文件位置，与-bootclasspath 相同。
- /usr/j2me_cldc/bin/common/api/classes 指 KVM/CLDC java 库的类文件路径。
- ：用于分开两个不同路径。
- tmpclasses 存放编译好的.class 文件路径，即待预验证的文件。
- -d 后面跟验证后输出文件路径。
- . 表示当前路径。
- TestKVM 待验证的类名文件，注意不要写成 TestKVM.class。

当然上面的写法不唯一，可以作为参考，读者可以根据各项参数说明，写成自己习惯的写法。编译完成检查当前目录下生成的文件。

```
# ls
```

在当前目录中生成了验证后的文件 TestKVM.class。执行下面的命令进行测试文件 TestKVM.class。

```
# kvm -classpath . TestKVM
```

执行结果：

```
Hello KVM!
```

执行结果说明文件 TestKVM.class 预验证成功。

### 19.8.5　移植

移植的文件主要有两个，一个是预验证好的文件 TestKVM.class 和交叉编译好的可执行文件 KVM。本次测试采用 NFS 方式进行测试，也回顾一下 NFS。下面提示一下 NFS 的关键步骤，如果有不熟悉的读者，可以重新复习第 2 章的 NFS 挂载部分。

（1）复制文件到虚拟机的 NFS 相关目录下。

```
# cp /usr/j2me_cldc/kvm/VmUnix/build/kvm /home/nfs/usr/
                                           //复制 KVM 到 USR 目录下
# cp /usr/JAVA/TestKVM.class /home/nfs/usr/
                                           //复制 TestKVM.class 到 USR 目录下
```

（2）检查网络是否连接好，连接方式是否为网桥方式。

（3）检验虚拟机和 ARM 板连接状态。为确保虚拟机和 ARM 板之间为可达，可以使用 ping 命令验证。在设置 IP 时，将两者的 IP 段设置在一个 IP 段，ping 的结果如图 19.7 所示。

```
# ifconfig                      //用于检查两侧的 IP 地址是否在一个网段
# ping 192.168.1.123            //在 ARM 板上 ping 主机
```

（4）检查 NFS 配置。

```
#vi /etc/exports
```

本机的配置如下：

```
#/home/nfs 192.168.1.*(rw,sync,no_root_squash)
/usr/local/linphone/linphone_arm 192.168.1.*(rw,sync,no_root_squash)
```

将第 1 行注释取消，并注释第 2 行，读者可以针对自己的情况设置，这里主要是提醒读者对于 NFS 的步骤设置。修改配置后应该使用 exportfs –rv 命令使配置重新生效，如图 19.7 所示。

```
# exportfs -rv
```

（5）启动 portmap 服务和 NFS 服务。

```
# service portmap restart
停止 portmap:                         [确定]
启动 portmap:                         [确定]
# service nfs restart
关闭 NFS mountd:                      [确定]
```

```
关闭 NFS 守护进程:                        [确定]
关闭 NFS quotas:                          [确定]
关闭 NFS 服务:                            [确定]
启动 NFS 服务:                            [确定]
关掉 NFS 配额:                            [确定]
启动 NFS 守护进程:                        [确定]
启动 NFS mountd:                          [确定]
```

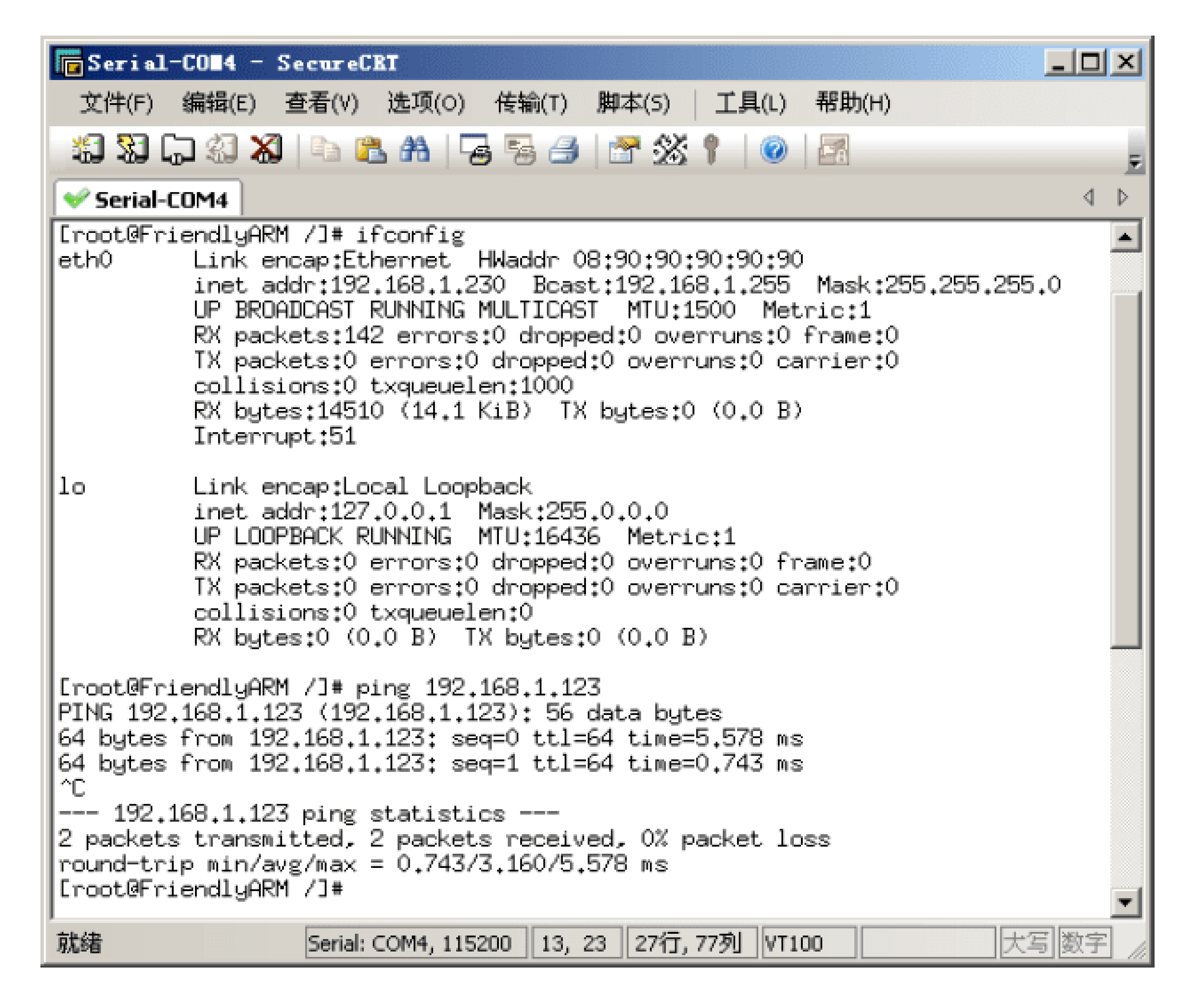



图 19.7　虚拟机和 ARM 可以 ping 通

上面的启动过程还要注意启动顺序，后面是正确启动的信息，如果不能正确启动，请参考第 2 章的说明。

（6）关闭防火墙。关闭的防火墙包括 Windows 主机防火墙和 Linux 虚拟机的防火墙。

（7）挂载 NFS。

```
# mount -o nolock -t nfs 192.168.1.123:/home/nfs /mnt
```

（8）移植动态链接库。移植动态链接库前，保证编译内核和文件系统的编译器与编译 KVM 编译器为相同版本的编译器。在目录/usr/j2me_cldc/bin/linux 下编译的 KVM 是运行在 X86 平台上的，查看其依赖的库文件。将库文件移植到/lib 目录下。

```
# ldd kvm
       linux-gate.so.1 =>  (0x00397000)
       libm.so.6 => /lib/libm.so.6 (0x446c7000)
       libnsl.so.1 => /lib/libnsl.so.1 (0x43c37000)
       libc.so.6 => /lib/libc.so.6 (0x44588000)
       /lib/ld-linux.so.2 (0x43bb9000)
```

将交叉编译器下的这些动态链接库移植到开发板的/lib 目录下。

（9）运行测试。执行测试的时候最好将 KVM 与 TestKVM 放置在同一个目录下，避免因路径引起的装载时名字错误而导致异常现象。

```
# kvm TestKVM
```

- 正确的显示结果为 Hello KVM!
- 如果出现找不到 KVM，说明没有正确的库文件；
- 如果出现权限不够，则需要修改 KVM 权限。

移植时为了省去移植动态链接库的麻烦，还可以使用静态编译的方式，静态编译时需修改目录/usr/j2me_cldc/kvm/VmUnix/build 下的 Makefile 文件。

```
kvm$(j)$(g): obj$j$g/ fp_obj$j$g/ $(CLEANUPXPM) $(OBJFILES) $(FP_OBJFILES)
	@echo "Linking ... $@"
	@$(CC) $(OBJFILES) $(FP_OBJFILES) -o $@ $(LIBS)
```

添加编译参数-static。

```
kvm$(j)$(g): obj$j$g/ fp_obj$j$g/ $(CLEANUPXPM) $(OBJFILES) $(FP_OBJFILES)
	@echo "Linking ... $@"
	@$(CC) -static $(OBJFILES) $(FP_OBJFILES) -o $@ $(LIBS)
```

修改完成后，回到/usr/j2me_cldc/build/linux 目录下进行编译，静态编译的可执行文件比较大，编译完后可以通过 file 命令进行查看：

```
# file kvm
kvm: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.0.0,
statically linked, for GNU/Linux 2.0.0, not stripped
```

## 19.9　小　　结

Java 语言有很多优点，包括完全面向对象、平台可移植性、不需要程序员进行内存回收等，相对于 C 和 C++来说更安全。而且 Java 有专门针对嵌入式平台的分支 j2me，目前移动终端、PDA 上越来越流行 Java 程序。移植完成 Java 虚拟机后，就可以将一些在 PC 上的 Java 应用程序移植到嵌入式环境中，移植这些 Java 应用程序比 C 和 C++应用程序简单得多。

# 第 20 章　VoIP 技术与 Linphone 编译

Linphone 是一个运行于 Linux 下的小型万维网电话应用程序 VoIP（Voice over Internet Protocol）。它允许用户通过因特网来进行双方通话和视频。不需要特定的硬件项目：安装了声卡的标准工作站、麦克风和扬声器或耳机，只要具备以上硬件即可开始使用 Linphone 进行语音、文本和视频通信。本章重点介绍 VoIP 的基础知识和 Linphone 涉及的协议介绍，同时还介绍 Linphone 编译和移植方法。

## 20.1　VoIP 介绍

VoIP 是利用 IP 网络传送语音、传真、视频和数据等业务，为用户节省通信的一种通信设备。它主要适合有分支机构的企业和集团用户，能给企业节省大量的国际、国内和郊区长途通信费用。下面将介绍其基本原理、优点、过程和实现方式。

### 20.1.1　VoIP 基本原理

VoIP 就是 IP 分组上承载话音，其基本原理是：通过语音压缩算法对语音数据进行压缩编码处理，然后把这些语音数据按 IP 等相关协议进行打包，经过 IP 网络把数据包传输到接收地，再把这些语音数据包串起来，经过解码解压处理后，恢复成原来的语音信号，从而达到由 IP 网络传送语音的目的。现有的简单 VoIP 系统结构如图 20.1 所示。

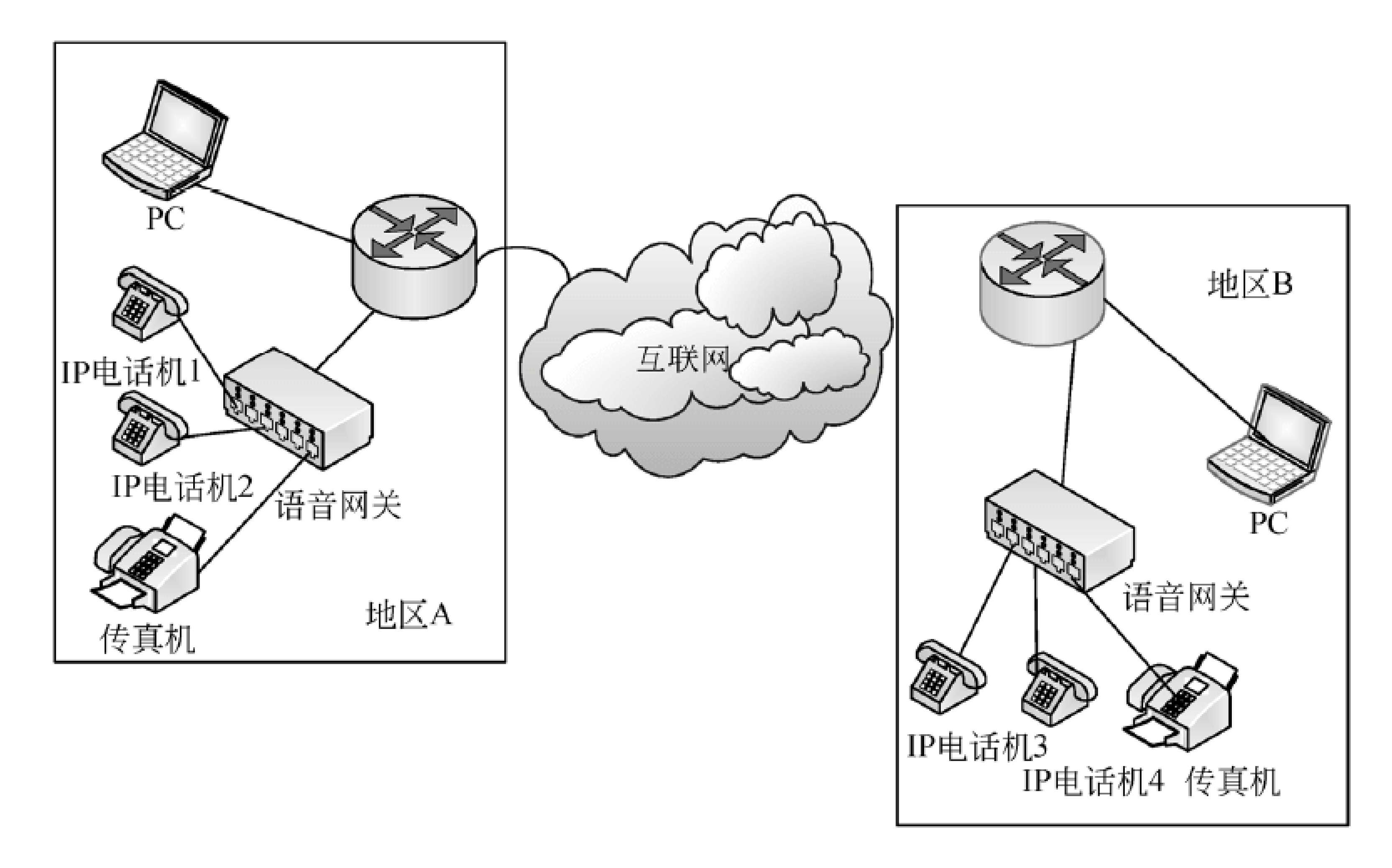



图 20.1　VoIP 的系统结构

## 20.1.2　VoIP 的基本传输过程

VoIP 实现通信双方或多方通信经过了下面一系列的过程。

（1）模拟语音转换数据。

（2）原数据转换为 IP 包。

（3）传送 IP 包。

（4）IP 包转换为数据。

（5）数字语音转换为模拟语音。

VoIP 的语音传输过程，如图 20.2 所示。

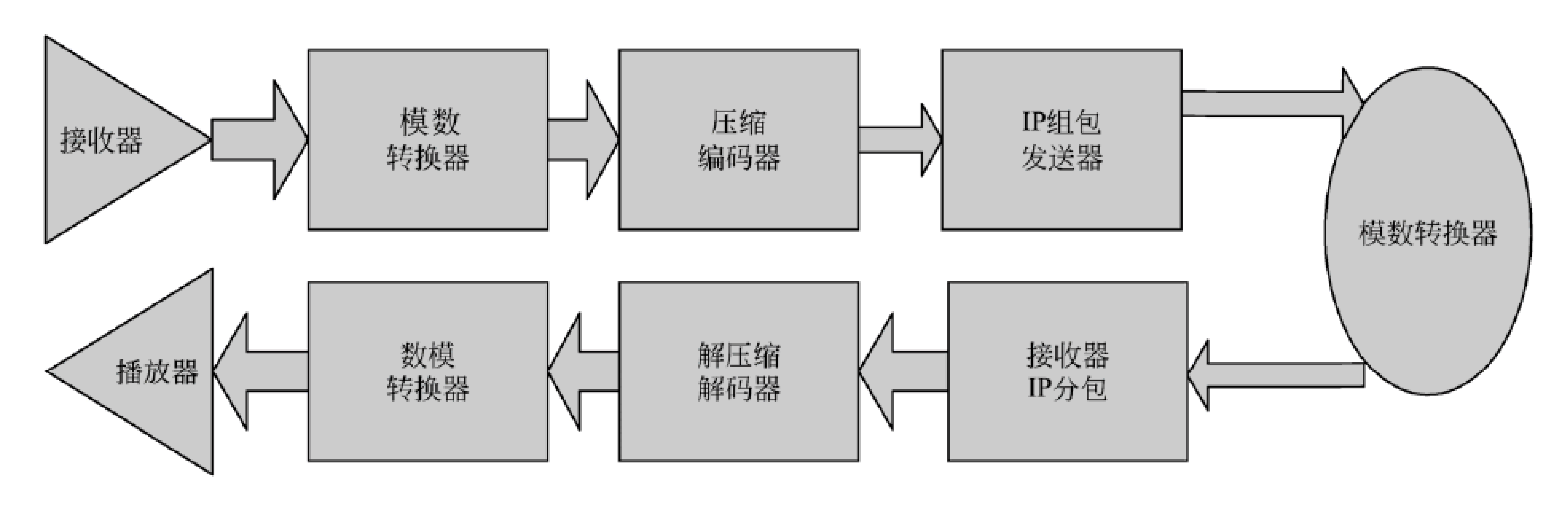



图 20.2　VoIP 语音传输图

## 20.1.3　VoIP 的优势

VoIP 相比传统电话具有明显的优势，其主要优势体现如下：

- 低廉的通信资费；
- 地理无关和号码漫游；
- 将语音网络和数据网络有机结合；
- 扩展了传统电话的功能，如视频通话、多方通话、视频会议、统一消息、数据存储转发、传真、流媒体等；
- 更多的应用和服务，如交互式电子商务、企业传真、多媒体视讯、智能代理等；
- 低廉的网络租赁维护费用。

## 20.1.4　VoIP 的实现方式

VoIP 主要有 4 种实现方式：电话机到电话机、电话机到 PC、PC 到电话机和 PC 到 PC。最初 VoIP 实现方式主要是 PC 到 PC。它利用 IP 地址进行呼叫，通过语音压缩、组包传送方式，实现互联网上 PC 机间的实时话音传送。在 PC 到 PC 的实现方式中，话音压缩、编解码和组包均通过 PC 上的处理器、声卡、网卡等硬件资源完成，这种方式和公用电话通信之间存在较大的差别，且限定在因特网内，所以有较大的局限性。

电话机到电话机，即普通电话经过电话交换机连到 IP 电话网关，使用电话号码穿越 IP

网进行呼叫，主叫端网关鉴别主叫用户，翻译电话号码 / 网关 IP 地址，发起 IP 电话呼叫，连接到被叫端网关，并完成话音编码和组包，接收端网关实现分包、解码和连接被叫。

对于电话机到 PC 或是 PC 到电话的情况，是由网关来完成 IP 地址和电话号码的对应和翻译，以及话音编解码和组包。

### 20.1.5　VoIP 的关键技术

IP 网络目的是用来传输数据业务，采用的是尽力而为的、无连接的技术。因此，它没有服务质量保证，存在分组丢失、失序到达和时延抖动等情况。而话音业务属于实时业务，对时序、时延等有严格的要求，必须采取其他服务质量保证业务质量。VoIP 的关键技术包括信令技术、编码技术、实时传输技术、服务质量保证技术及网络传输技术等。

- 信令技术：主要包括 ITU-T 的 H.323 和 IETF 会话初始化协议 SIP（Session Initation Protocol）两套标准体系，还涉及进行实时同步连续媒体流传输控制的实时流协议 TRSP。
- 编码技术：包括流行的 G.723.1、G.729、G.729A 话音压缩编码算法和 MPEG-II 多媒体压缩技术。
- 实时传输技术：主要采用实时传输协议 RTP。
- 服务质量保证技术：运用资源预留协议 RSVP 和用于业务质量监控的实时传输控制协议 RTCP 来解决网络拥塞，保证通话质量。
- 网络传输技术：主要是面向连接的 TCP 协议和无连接的 UDP 协议。

后面将结合代码介绍部分关键技术和协议，讲解它们的实现过程和原理。

## 20.2　oSIP 协议概述

SIP 协议会话控制协议，用来建立、修改和终止多媒体会话。oSIP 协议是用标准 C 编写的一个 SIP 协议栈，在编译 Linphone 时采用了支持 oSIP 协议的源码包 libosip2-3.3.0.tar.gz 和 libeXosip2-3.1.0.tar.gz，编译后分别得到 osip 的库和 eXosip 库文件，它们是 oSIP 的协议库和 oSIP 协议扩展库文件。

oSIP 协议栈主要分为 3 大部分：状态机模块、解析器模块和工具模块。3 个模块的功能如下所示。

- 状态机模块功能：记录某个事务（注册过程、呼叫过程）的状态，并在特定的状态下触发某个相应的事件或回调函数。
- 解析器模块功能：该模块主要完成对 SIP 消息的解析、SDP 消息的解析、URL 消息的解析。
- 工具模块功能：提供 SDP 等处理工具。

oSIP 协议栈的状态机又分为 4 种类型，分别如下：

- ICT：Invite Client (outgoing) Transaction；
- NICT：Non-Invite Client (outgoing) Transaction；
- IST：Invite Server (incoming) Transaction；

❑ NIST：Non-Invite Server (incoming) Transaction。

下面通过对 oSIP 几个重要部分的了解来进一步理解 oSIP，并通过对 oSIP 状态机、oSIP 解析器、oSIP 事务层等源码分析理解 oSIP 的实现细节。

# 20.3　oSIP 状态机

前面介绍的 4 种状态机 ICT、NICT、IST 和 NIST 在 osip.h 文件中的 state_t 结构体中定义，每种状态机都包含 5 种状态：准备呼叫（PRE_CALLING）、呼叫（CALLING）、处理（PROCEEDING）、完成（COMPLETED）、终止（TERMINATED），该结构体具体定义如下：

```
typedef enum _state_t
  {
    /*ICT 状态机的各个状态*/
    ICT_PRE_CALLING,
    ICT_CALLING,
    ICT_PROCEEDING,
    ICT_COMPLETED,
    ICT_TERMINATED,

    /*IST 状态机的各个状态*/
    IST_PRE_PROCEEDING,
    IST_PROCEEDING,
    IST_COMPLETED,
    IST_CONFIRMED,
    IST_TERMINATED,

    /*NICT 状态机的各个状态*/
    NICT_PRE_TRYING,
    NICT_TRYING,
    NICT_PROCEEDING,
    NICT_COMPLETED,
    NICT_TERMINATED,

    /*NIST 状态机的各个状态*/
    NIST_PRE_TRYING,
    NIST_TRYING,
    NIST_PROCEEDING,
    NIST_COMPLETED,
    NIST_TERMINATED,
  }
  state_t;
```

状态机一般为一个程序的核心部分，理解了状态机就能清楚地了解整个程序的流程。下面分别对这 4 种状态机进行图示分析与代码分析，读者先获得最新的 libosip2 压缩包，解压后熟悉 src 下面的文件。

## 20.3.1　ICT（Invite Client (outgoing) Transaction）状态机

在 libosip 源码目录下的 ICT 状态机文件为 ict_fsm.c，数组 ict_transition[11]给定了各个

状态接收事件的各种事务处理情况。

```
transition_t ict_transition[11] =
 {
   {
    ICT_PRE_CALLING,            //初始状态为 vICT_PRE_CALLING
    SND_REQINVITE,              //当接收到 SND_REQINVITE 事件时
    (void (*)(void *, void *)) &ict_snd_invite,
                          //调用处理事件函数 ict_snd_invite()
    &ict_transition[1], NULL
   }
   ,
   {
    ICT_CALLING,                //初始状态为 ICT_CALLING
    TIMEOUT_A,                  //当接收到 TIMEOUT_A 事件时
    (void (*)(void *, void *)) &osip_ict_timeout_a_event,
                                //调用处理函数 osip_ict_timeout_a_event()
    &ict transition[2], NULL
   }
   ,
   {
    ICT_CALLING,                //初始状态为 ICT_CALLING
    TIMEOUT_B,                  //当接收到 TIMEOUT_B 事件时
    (void (*)(void *, void *)) &osip_ict_timeout_b_event,
                          //调用处理函数 osip_ict_timeout_b_event()
    &ict_transition[3], NULL
   }
   ,
   { ICT_CALLING,               //初始状态为 ICT_CALLING
    RCV_STATUS_1XX,             //当接收到 RCV_STATUS_1XX 事件时
    (void (*)(void *, void *)) &ict_rcv_1xx,
                          //调用处理函数 ict_rcv_1xx ()
    &ict_transition[4], NULL
   }
   ,
   { ICT_CALLING,               //初始状态为 ICT_CALLING
    RCV_STATUS_2XX,             //当接收到 RCV_STATUS_2XX 事件时
    (void (*)(void *, void *)) &ict_rcv_2xx,
                          //调用处理函数 ict_rcv_2xx ()
    &ict_transition[5], NULL
   }
   ,
   { ICT_CALLING,               //初始状态为 ICT_CALLING
    RCV_STATUS_3456XX,          //当接收到 RCV_STATUS_3456XX 事件时
    (void (*)(void *, void *)) &ict_rcv_3456xx,
                          //调用处理函数 ict_rcv_3456xx ()
    &ict_transition[6], NULL
   }
   ,
   { ICT PROCEEDING,            //初始状态为 ICT PROCEEDING
    RCV_STATUS_1XX,             //当接收到 RCV_STATUS_1XX 事件时
    (void (*)(void *, void *)) &ict_rcv_1xx,
                          //调用处理函数 ict_rcv_1xx ()
    &ict transition[7], NULL
   }
   ,
   { ICT_PROCEEDING,            //初始状态为 ICT_PROCEEDING
```

```
      RCV_STATUS_2XX,              //当接收到 RCV_STATUS_2XX 事件时
      (void (*)(void *, void *)) &ict_rcv_2xx,
                               //调用处理函数 ict_rcv_2xx ()
      &ict transition[8], NULL
    }
    ,
    { ICT_PROCEEDING,              //初始状态为 ICT_PROCEEDING
      RCV_STATUS_3456XX,           //当接收到 RCV_STATUS_3456XX 事件时
      (void (*)(void *, void *)) &ict_rcv_3456xx,
                               //调用处理函数 ict_rcv_3456xx ()
      &ict transition[9], NULL
    }
    ,
    { ICT_COMPLETED,               //初始状态为 ICT_COMPLETED
      RCV_STATUS_3456XX,           //当接收到 RCV_STATUS_3456XX 事件时
      (void (*)(void *, void *)) &ict_retransmit_ack,
                               //调用处理函数 ict_retransmit_ack()
      &ict transition[10], NULL
    }
    ,
    { ICT_COMPLETED,               //初始状态为 ICT_COMPLETED
      TIMEOUT_D,                   //当接收到 TIMEOUT_D 事件时
      (void (*)(void *, void *)) &osip_ict_timeout_d_event,
                               //调用处理函数 osip ict timeout d event()
      NULL, NULL
    }
  };
```

### 1．错误事务处理函数ict_handle_transport_error()

任何状态接收到错误事件时，都会产生一个回调函数__osip_transport_error_callback()，然后进入 ICT_TERMINATED 状态，然后调用回调函数__osip_kill_transaction_callback()退出状态机。

```
static void ict_handle_transport_error (osip_transaction_t * ict, int err)
{
  /*调用回调函数__osip_transport_error_callback()*/
    osip transport error callback (OSIP ICT TRANSPORT ERROR, ict, err);
  /*进入状态机的终结状态*/
    osip transaction set state (ict, ICT TERMINATED);
  /*调用回调函数__osip_kill_transaction_callback()退出状态机*/
    osip kill transaction callback (OSIP ICT KILL TRANSACTION, ict);
  /* TODO: MUST BE DELETED NOW */
}
```

由上面错误处理函数 ict_handle_transport_error()所表示的状态迁移关系，可以得到状态图 20.3 中粗实线条部分。

### 2．发送邀请函数ict_snd_invite()

发送邀请函数 ict_snd_invite()，是预呼叫状态（ICT_PRE_CALLING）下的处理函数，当发送消息成功时调用回调函数__osip_message_callback()并进入呼叫状态（ICT_CALLING）。

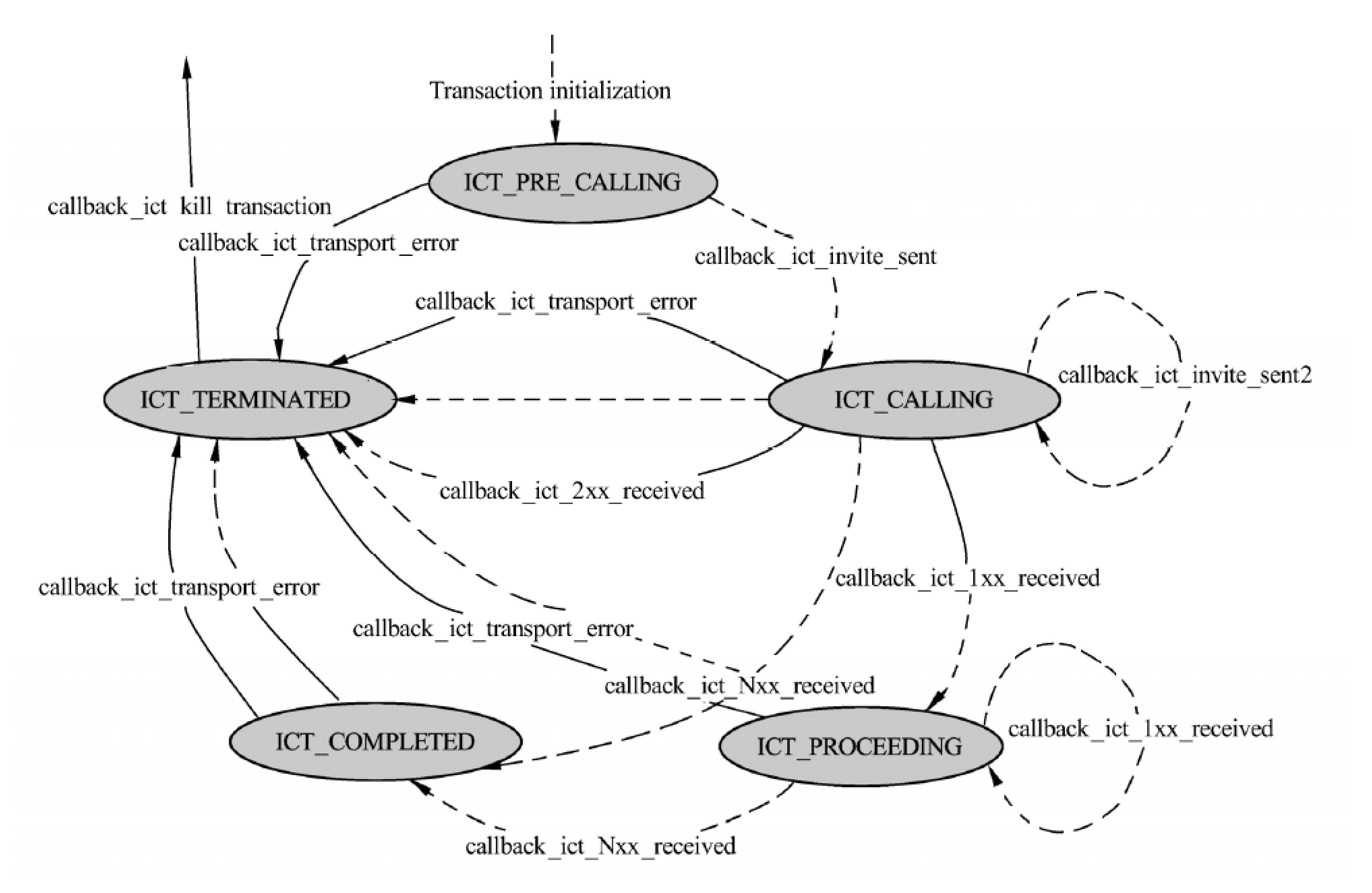



图 20.3　ICT 状态机的错误事务处理

```
void ict_snd_invite (osip_transaction_t * ict, osip_event_t * evt)
{
  int i;
  osip_t *osip = (osip_t *) ict->config;

  /* Here we have ict->orig_request == NULL */
  ict->orig_request = evt->sip;
  /*发送消息成功则返回 0，否则返回非 0*/
  i = osip->cb_send_message (ict, evt->sip, ict->ict_context->destination,
                            ict->ict_context->port, ict->out_socket);

  if (i == 0)
  {
      /*成功则调用回调函数*/
      __osip_message_callback (OSIP_ICT_INVITE_SENT, ict, ict->orig_
      request);
      /*进入呼叫状态*/
      __osip_transaction_set_state (ict, ICT_CALLING);
  } else
    {
      /*邀请失败则进行错误处理*/
      ict handle transport error (ict, i);
    }
}
```

上面的发送邀请函数 ict_snd_invite()所表示的状态迁移关系，在图 20.4 中用粗实线体现出来。

### 3．应答函数ict_rcv_1xx()

根据数组 ict_transition[11] 的定义，调用函数 ict_rcv_1xx() 的初始状态有

ICT_PROCEEDING 和 ICT_CALLING 两种情况，而根据函数 ict_rcv_1xx()处理的情况看，结束状态均为 ICT_PROCEEDING。

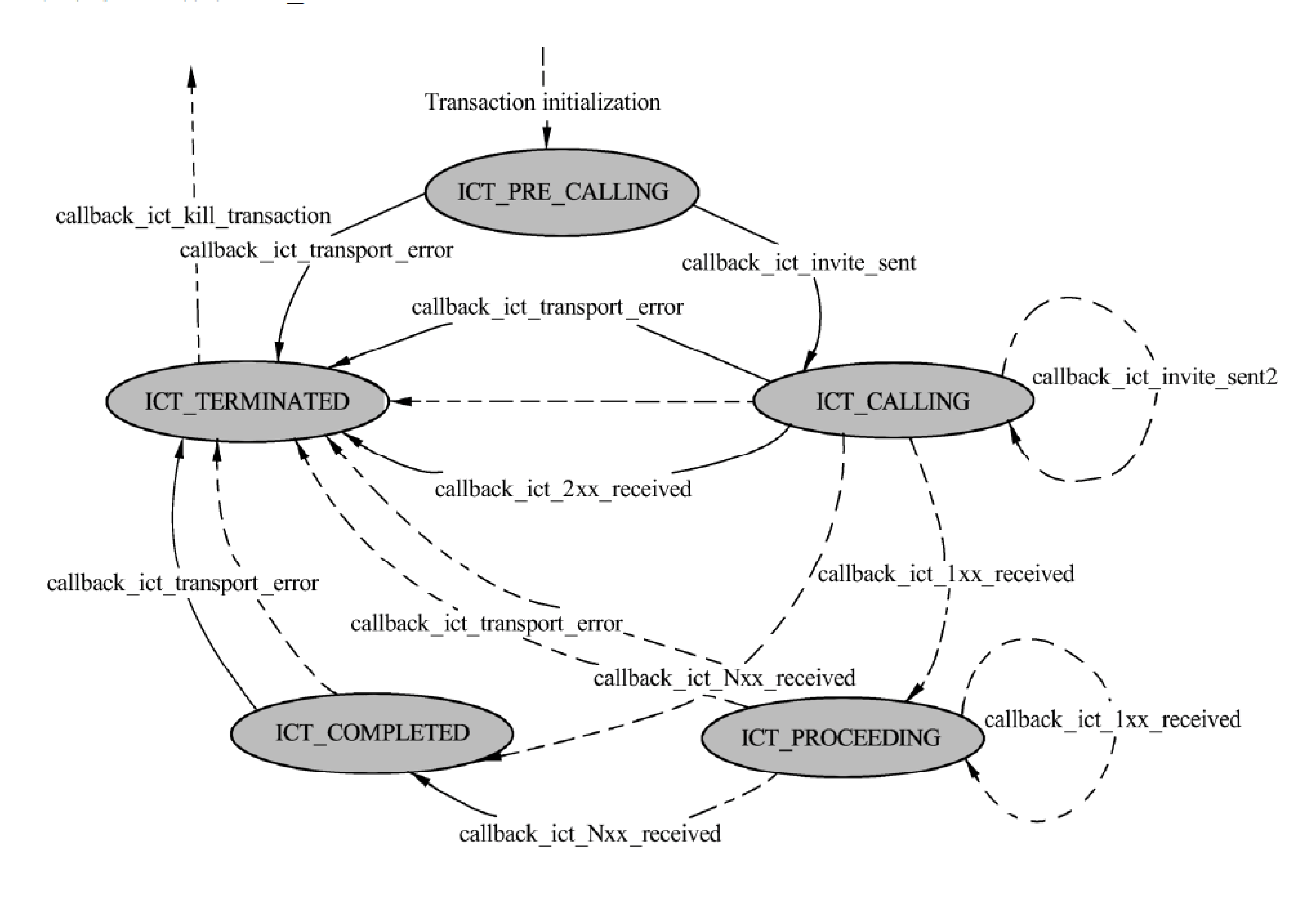



图 20.4　PRE_CALLING 发送邀请事件

```
void ict_rcv_1xx (osip_transaction_t * ict, osip_event_t * evt)
{
  /* leave this answer to the core application */

  if (ict->last response != NULL)
    {
      osip message free (ict->last response);
    }
  ict->last_response = evt->sip;
  /*调用回调函数*/
  __osip_message_callback (OSIP_ICT_STATUS_1XX_RECEIVED, ict, evt->sip);
  /*函数的出口状态均为 ICT_PROCEEDING*/
  __osip_transaction_set_state (ict, ICT_PROCEEDING);
}
```

应答函数 ict_rcv_1xx()和应答函数 ict_rcv_2xx()一起在状态转移图中体现的结果，如图 20.5 所示。

#### 4．应答函数ict_rcv_2xx()

根据数组 ict_transition[11]的定义，调用函数 ict_rcv_2xx()的初始状态有 ICT_PROCEEDING 和 ICT_CALLING 两种状态，而根据函数 ict_rcv_2xx()处理的情况看结束状态均为 ICT_TERMINATED。如图 20.5 所示。

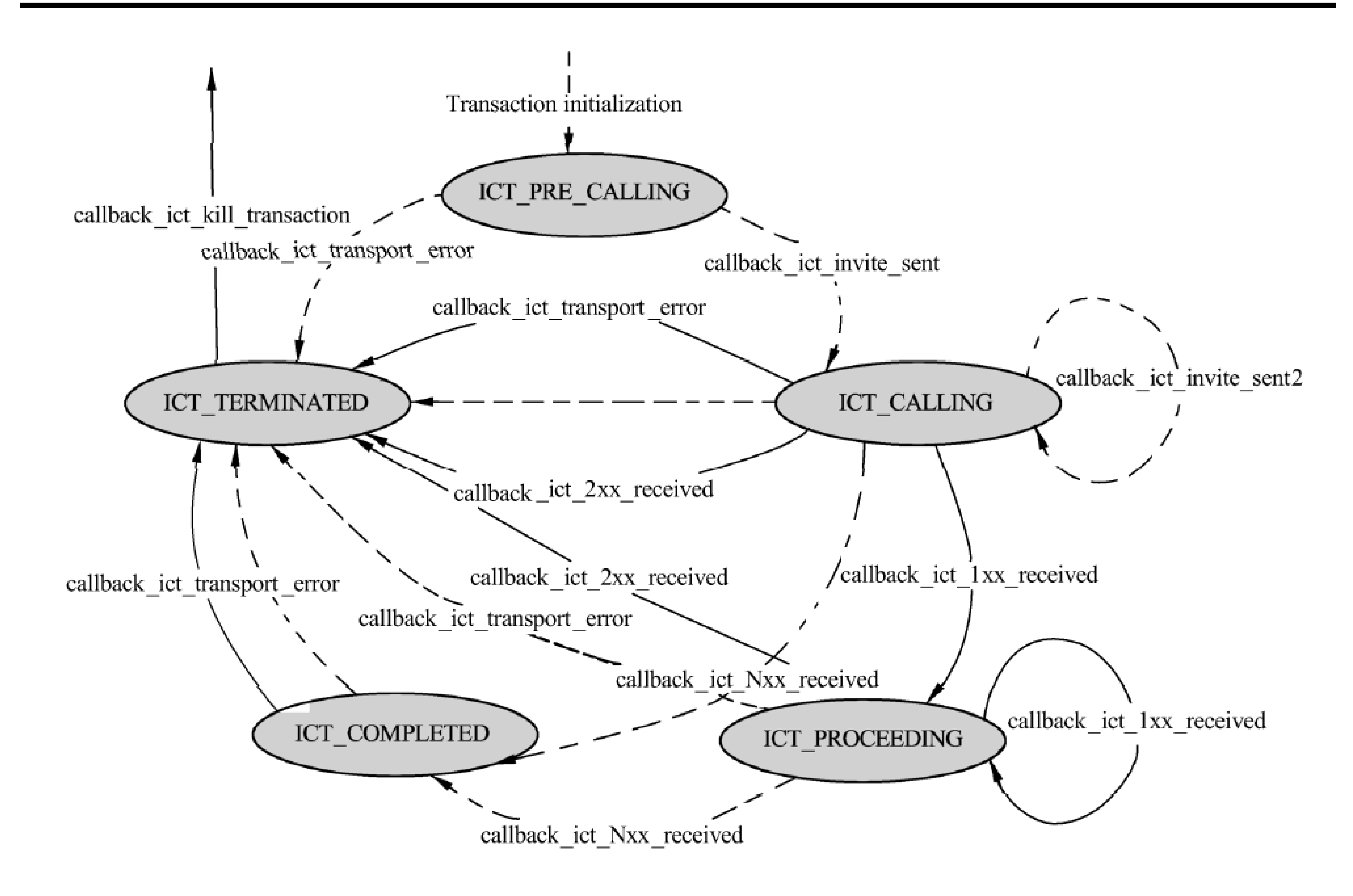



图 20.5　ict_rcv_1xx()与 ict_rcv_2xx()处理

```
void ict_rcv_2xx (osip_transaction_t * ict, osip_event_t * evt)
{
  /*leave this answer to the core application*/
  if (ict->last response != NULL)
    {
      osip_message_free (ict->last_response);
    }
  ict->last response = evt->sip;
/*调用回调函数*/
  __osip_message_callback (OSIP_ICT_STATUS_2XX_RECEIVED, ict, evt->sip);
/*进入 ICT_TERMINATED 状态*/
  __osip_transaction_set_state (ict, ICT_TERMINATED);
/*调用回调函数*/
  __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION, ict);
}
```

### 5．应答函数ict_rcv_3456xx()

根据数组 ict_transition[11]的定义，调用应答函数 ict_rcv_3456xx()的初始状态有ICT_PROCEEDING、ICT_COMPLETED 和 ICT_CALLING 这 3 种状态，函数 ict_rcv_3456xx()处理的结果最终状态为 ICT_COMPLETED。

```
void ict_rcv_3456xx (osip_transaction_t * ict, osip_event_t * evt)
{
  osip_route_t *route;
  int i;
  osip_t *osip = (osip_t *) ict->config;
  /*leave this answer to the core application*/
```

```
  if (ict->last_response != NULL)
    osip_message_free (ict->last_response);
  ict->last response = evt->sip;
  if (ict->state != ICT_COMPLETED)      /* not a retransmission */
    {
      /* automatic handling of ack! */
      osip_message_t *ack = ict_create_ack (ict, evt->sip);
      ict->ack = ack;
      if (ict->ack == NULL)
        {
          /*初始状态不是 ICT_COMPLETED ，根据数组 ict_transition[11]的定义是状态
          ICT_CALLING 和 ICT_PROCEEDING 时，直接跳转到 ICT_TERMINATED 状态*/
          __osip_transaction_set_state (ict, ICT_TERMINATED);
          /*调用回调函数*/
          __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION,
          ict);
          return;
        }
/*省略一些细节，为了更容易看出状态机跳转的主线*/
      i = osip->cb send message (ict, ack, ict->ict context->destination,
                              ict->ict_context->port, ict->out_socket);
      if (i != 0)
        {
          /*状态 ICT_CALLING 和 ICT_PROCEEDING 发送消息错误返回时，调用错误处理函数
          ict_handle_transport_error ()*/
          ict_handle_transport_error (ict, i);
          return;
        }
      if (MSG IS STATUS 3XX (evt->sip))
        /*调用回调函数*/
          osip message callback (OSIP ICT STATUS 3XX RECEIVED, ict, evt->
        sip);
      else if (MSG_IS_STATUS_4XX (evt->sip))
        /*调用回调函数*/
        __osip_message_callback (OSIP_ICT_STATUS_4XX_RECEIVED, ict, evt->
        sip);
      else if (MSG_IS_STATUS_5XX (evt->sip))
        /*调用回调函数*/
        __osip_message_callback (OSIP_ICT_STATUS_5XX_RECEIVED, ict, evt->
        sip);
      Else
        /*调用回调函数*/
          osip message callback (OSIP ICT STATUS 6XX RECEIVED, ict, evt->
        sip);
        /*调用回调函数*/
        osip message callback (OSIP ICT ACK SENT, ict, evt->sip);
    }
  /* start timer D (length is set to MAX (64*DEFAULT_T1 or 32000) */
  osip_gettimeofday (&ict->ict_context->timer_d_start, NULL);
  add_gettimeofday (&ict->ict_context->timer_d_start, ict->ict_context->
  timer d length);
  /*处理完该函数，正常退出时结束状态为 ICT_COMPLETED*/
  __osip_transaction_set_state (ict, ICT_COMPLETED);
}
```

根据上面的函数分析得到下面粗实线所表示的状态转移图，如图 20.6 所示。

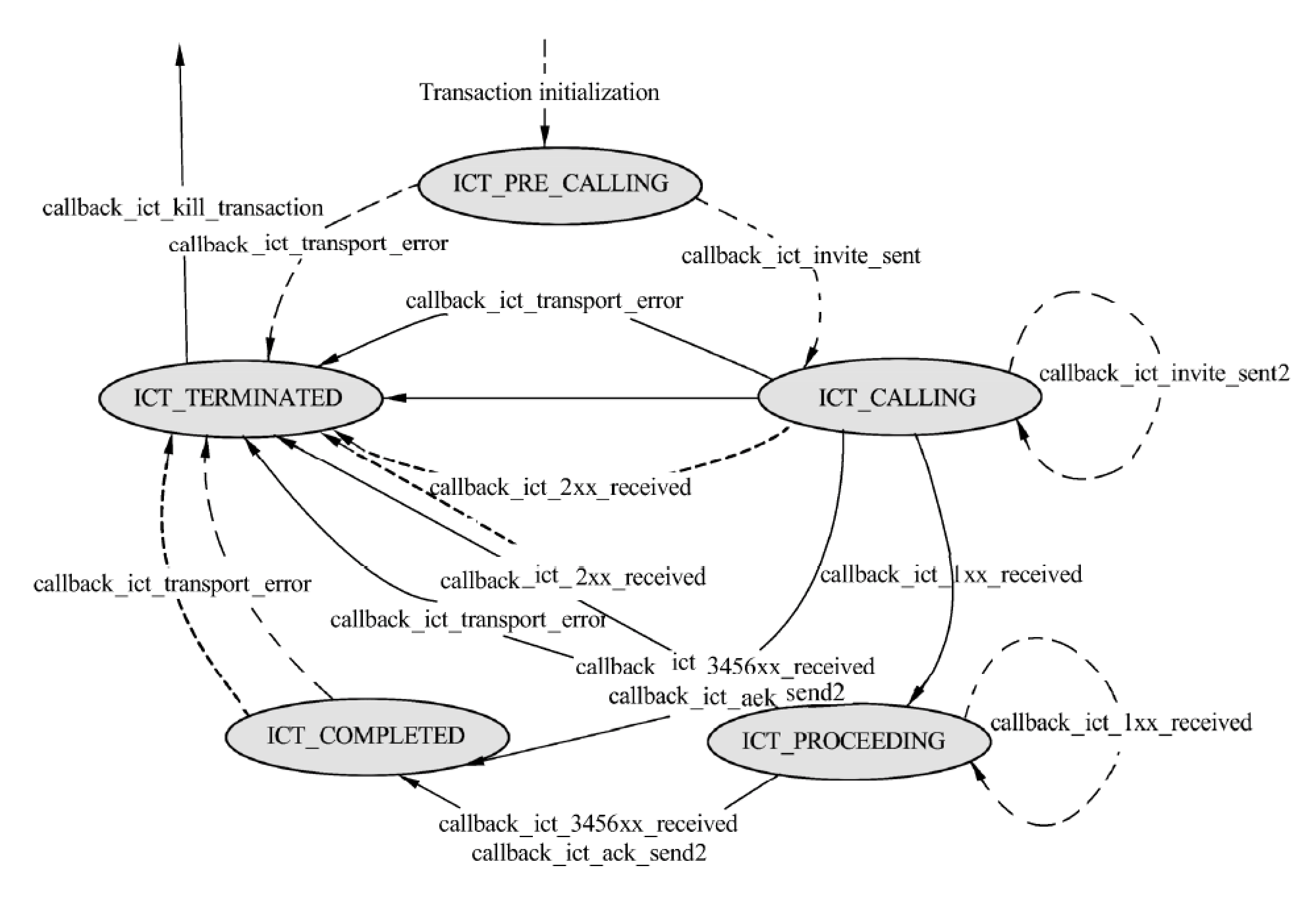



图 20.6　函数 ict_rcv_3456xx()表示的状态转移部分

### 6．超时事件

超时事件函数 osip_ict_timeout_b_event()和 osip_ict_timeout_d_event()涉及状态之间的跳转，接收事件 TIMEOUT_B 时，状态从 ICT_CALLING 跳转到 ICT_TERMINATED。接收超时事件 TIMEOUT_D 时，状态从 ICT_COMPLETED 跳转到 ICT_TERMINATED。两个事件处理函数如下，其状态转移图如图 20.7 粗线部分所示。

```
void osip_ict_timeout_b_event (osip_transaction_t * ict, osip_event_t * evt)
{
  ict->ict_context->timer_b_length = -1;
  ict->ict_context->timer_b_start.tv_sec = -1;
/*调用超时回调函数*/
  __osip_message_callback (OSIP_ICT_STATUS_TIMEOUT, ict, evt->sip);
/*设置状态为 ICT_TERMINATED*/
  __osip_transaction_set_state (ict, ICT_TERMINATED);
/*调用结束事务处理回调函数*/
  __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION, ict);
}
void osip_ict_timeout_d_event (osip_transaction_t * ict, osip_event_t * evt)
{
  ict->ict_context->timer_d_length = -1;
  ict->ict_context->timer_d_start.tv_sec = -1;
/*设置状态为 ICT_TERMINATED*/
  __osip_transaction_set_state (ict, ICT_TERMINATED);
/*调用结束事务处理回调函数*/
```

```
  __osip_kill_transaction_callback (OSIP_ICT_KILL_TRANSACTION, ict);
}
```

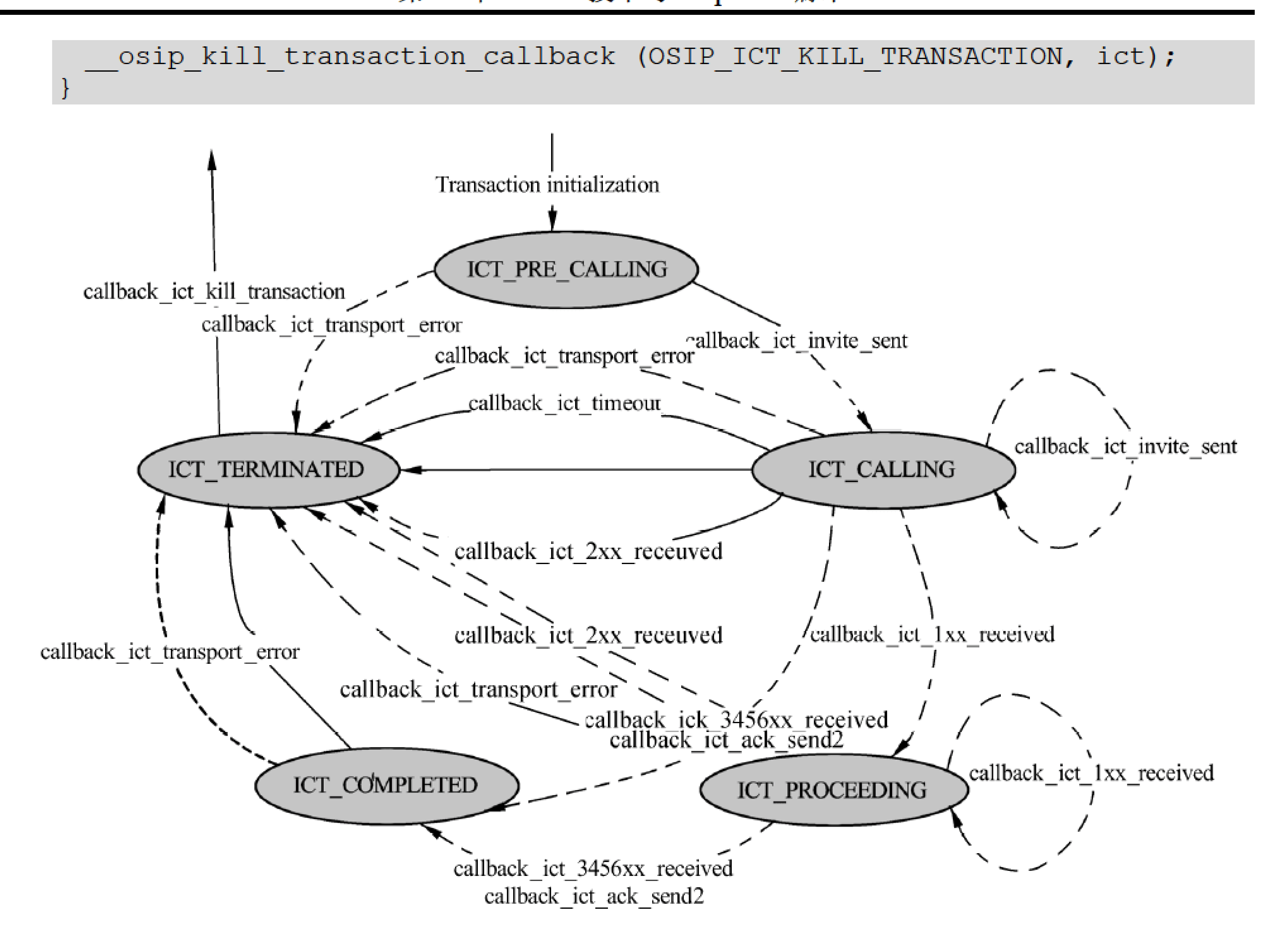



图 20.7　超时事件 TIMEOUT_B 和 TIMEOUT_B 所产生的状态转移

### 7．应答函数ict_retransmit_ack()

根据接收函数 ict_rcv_3456xx()中调用的 ACK_SENT 事件可知，ict_retransmit_ack()的起始状态为 ICT_COMPLETED，正常情况下的结束状态为 ICT_COMPLETED，出错时调用错误处理函数。由该函数得出的状态转移图如图 20.8 中粗实线所示。

```
void ict_retransmit_ack (osip_transaction_t * ict, osip_event_t * evt)
{
  int i;
  osip_t *osip = (osip_t *) ict->config;
  /*this could be another 3456xx ???*/
  /*we should make a new ACK and send it!!!*/
  /*调用回调函数*/
    osip message callback (OSIP ICT STATUS 3456XX RECEIVED AGAIN, ict,
  evt->sip);
  /*释放消息*/
  osip_message_free (evt->sip);

  i = osip->cb_send_message (ict, ict->ack, ict->ict_context->destination,
                              ict->ict_context->port, ict->out_socket);

  if (i == 0)
{
  /*调用回调函数再次应答发送*/
      __osip_message_callback (OSIP_ICT_ACK_SENT_AGAIN, ict, ict->ack);
      /*设置状态 ICT_COMPLETED*/
```

```
        __osip_transaction_set_state (ict, ICT_COMPLETED);
    } else
  {
    /*发送消息错误则进行错误处理*/
        ict handle transport error (ict, i);
      }
  }
```

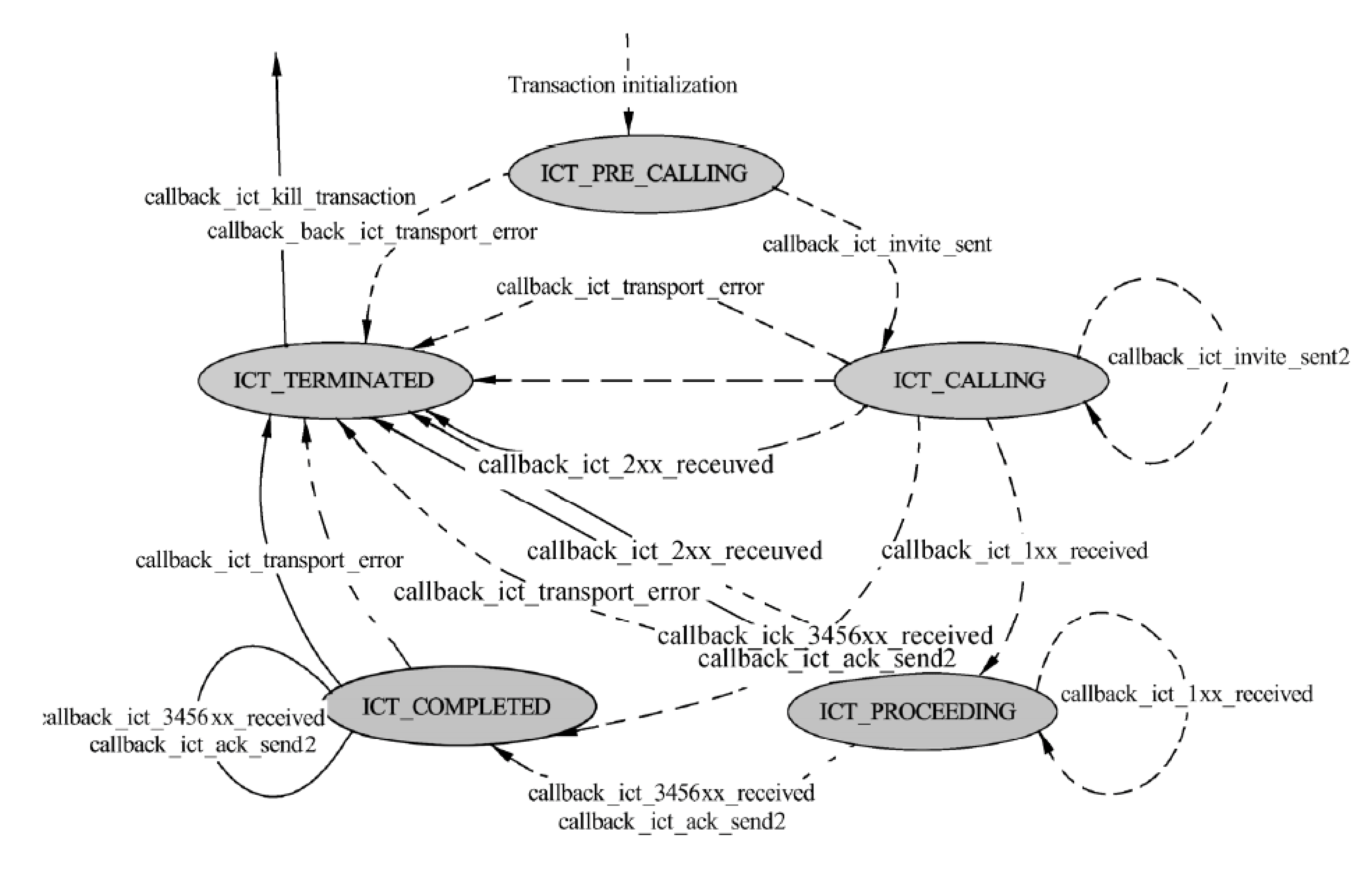



图 20.8　函数 ict_retransmit_ack()产生的状态转移

## 20.3.2　NICT（Non-Invite Client (outgoing) Transaction）状态机

NICT 状态机整个实现过程在 nict_fsm.c 文件中定义，在数组 transition_t nict_transition[12]中定义了状态机的初始状态及初始状态接收的事件，各个事件函数表示了状态机处理该事件的跳转情况，下面直接给出 NICT 的状态机跳转图 20.9，读者可以根据手中的代码自行推导和验证。NICT 状态机中和 ICT 状态机类似，具有以下 5 种状态：

- NICT_PRE_TRYING；
- NICT_TRYING；
- NICT_PROCEEDING；
- NICT_COMPLETED；
- NICT_TERMINATED。

状态之间的事件函数包括：

- 错误处理 nict_handle_transport_error()
- 发送请求 nict_snd_request()
- 超时事件 osip_nict_timeout_k_event()
- 6 类应答函数 nict_rcv_123456xx()

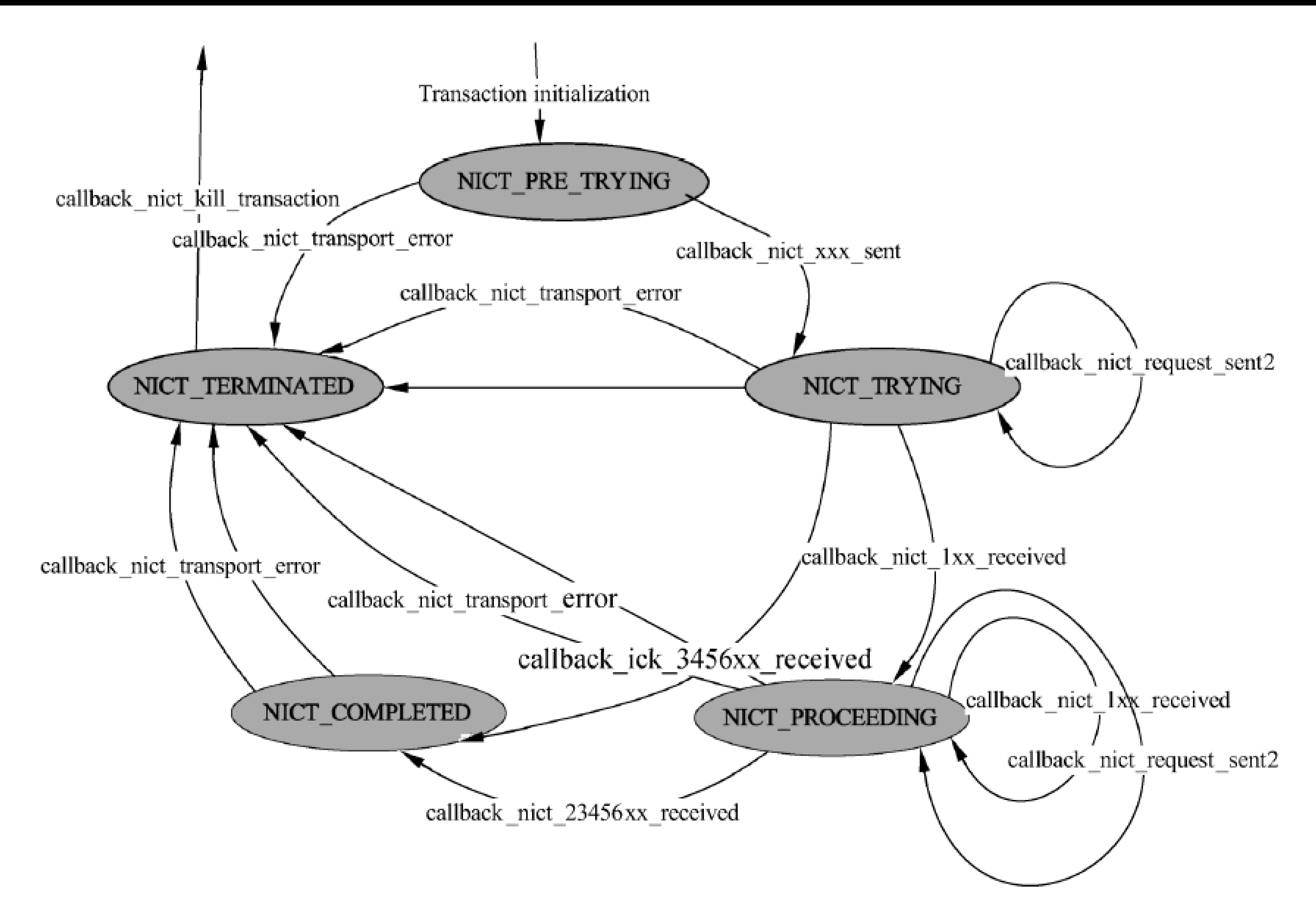



图 20.9　NICT 状态机的状态迁移图

### 20.3.3　IST（Invite Server (incoming) Transaction）状态机

IST 状态机整个实现过程在 ist_fsm.c 文件中定义，在数组 transition_t ist_transition[11] 中定义了状态机的初始状态及初始状态接收的事件，各个事件函数表示了状态机处理该事件的跳转情况，下面直接给出 IST 的状态机跳转图，如图 20.10 所示。

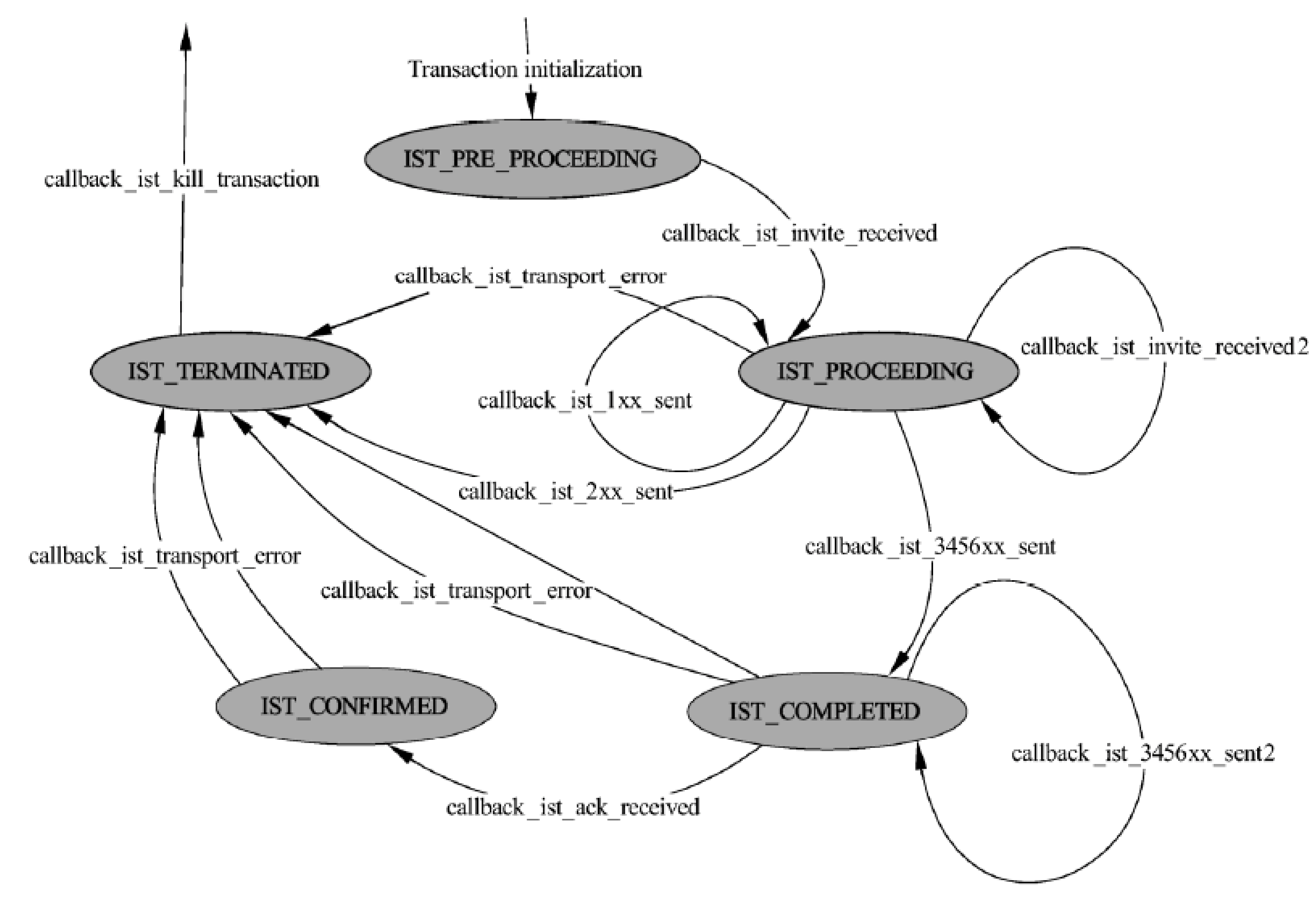



图 20.10　IST 状态机的状态迁移图

### 20.3.4　NIST（Non-Invite Server (incoming) Transaction）状态机

NIST 状态机整个实现过程在 nist_fsm.c 文件中定义，在数组 transition_t nist_transition[10]中定义了状态机的初始状态及初始状态接收的事件，各个事件函数表示了状态机处理该事件的跳转情况，下面直接给出 NIST 的状态机跳转图，如图 20.11 所示。

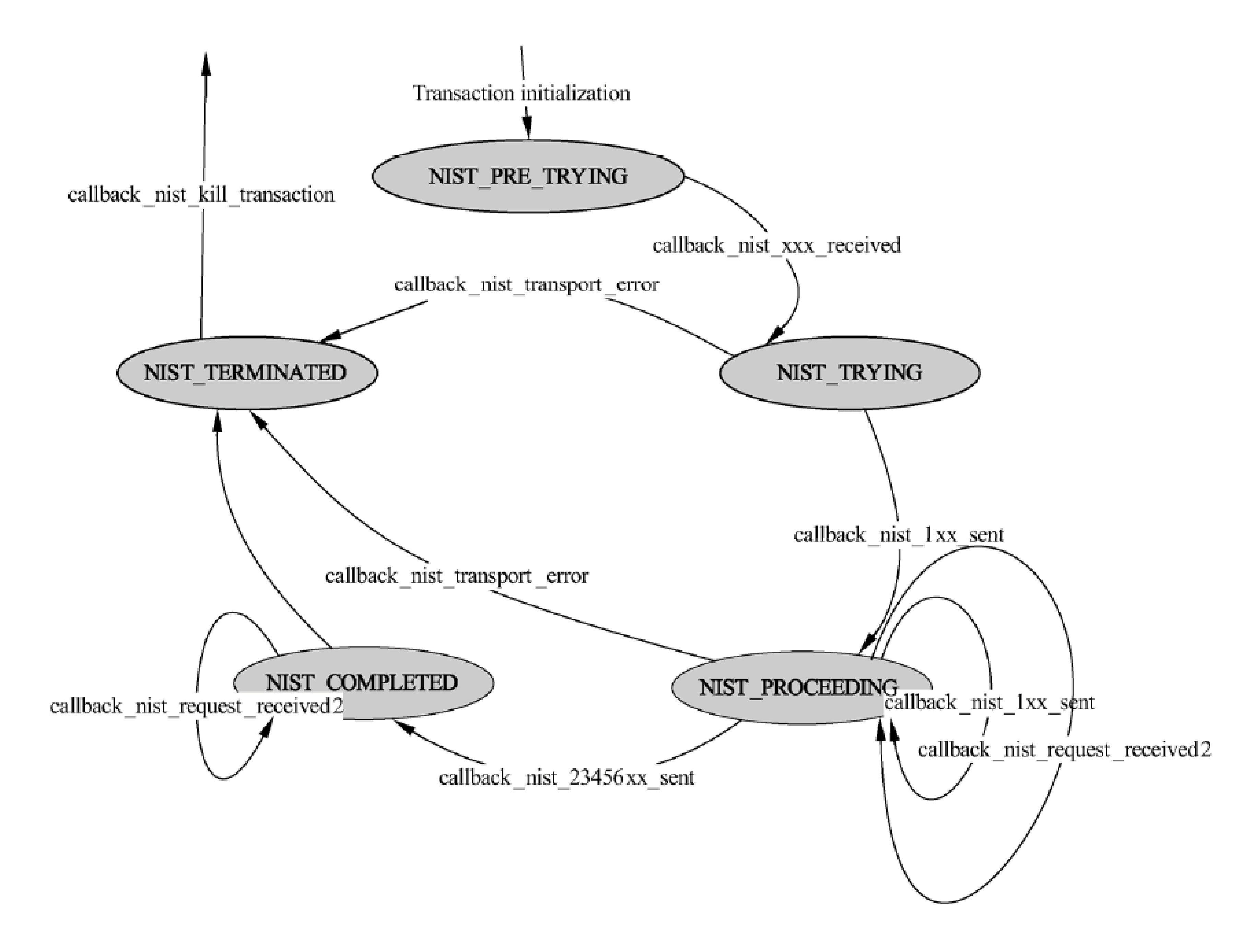



图 20.11　NIST 状态机的状态迁移图

## 20.4　oSIP 解析器

每种类型的 SIP 解析器所提供的 API 基本类似。通过函数 osip_xxx_init()创建解析类型 xxx，函数 osip_xxx_free()用于释放解析类型，函数 osip_xxx_parse()用于从字符串到解析类型的解析，函数 osip_xxx_to_str()用于从解析类型到字符串的解析，函数 osip_xxx_clone()用于从旧的解析类型复制到新的解析类型，函数 osip_xxx_set_header()用于设置解析类的头部，函数 osip_xxx_set_contenttype()用于设置解析类型内容。下面通过一个 body 类型实例进行说明。

### 20.4.1　初始化解析类型函数 osip_body_init()

函数 osip_body_init()为类型 body 分配空间，并初始化各个字段，并且将 body 的头部

也初始化为 0，整个初始化过程成功则返回 OSIP_SUCCESS。

```
Int osip_body_init (osip_body_t ** body)
{
  /*为 body 分配空间*/
  *body = (osip_body_t *) osip_malloc (sizeof (osip_body_t));
  /*分配失败则退出*/
  if (*body == NULL)
    return OSIP_NOMEM;
    /*为 body 的各个字段赋值*/
  (*body)->body = NULL;
  (*body)->content_type = NULL;
  (*body)->length = 0;
  /*为 body 的头部分配空间*/
  (*body)->headers = (osip_list_t *) osip_malloc (sizeof (osip_list_t));
  if ((*body)->headers == NULL)
    {
      osip_free (*body);
      *body = NULL;
      return OSIP_NOMEM;
}
/*初始化头部*/
  osip_list_init ((*body)->headers);
  return OSIP_SUCCESS;
}
```

## 20.4.2　释放函数 osip_body_free()

函数 osip_body_free()用于释放函数 osip_body_init()初始化的结构体 body。

```
void osip_body_free (osip_body_t * body)
{
  if (body == NULL)
    return;
  osip free (body->body);
  if (body->content_type != NULL)
{
/*释放内容类型*/
      osip_content_type_free (body->content_type);
    }
  /*移除 body 头部*/
  osip_list_special_free(body->headers, (void *(*)(void *)) &osip_header_
  free);
  /*释放头部占用空间*/
  osip_free (body->headers);
  /*释放 body 结构体占用的空间*/
  osip_free (body);
}
```

## 20.4.3　字符串到 body 类型转换函数 osip_body_parse()

函数 osip_body_parse()用于将字符串类型转换成 body 类型。将输入参数 start_of_body

中的字符串以内存复制的方式赋值给 body 的 body 字段。

```
Int osip_body_parse (osip_body_t * body, const char *start_of_body, size_t
length)
{
  /*转换前进行安全检查*/
  if (body == NULL)
    return OSIP_BADPARAMETER;
  if (start of body == NULL)
    return OSIP_BADPARAMETER;
  if (body->headers == NULL)
    return OSIP BADPARAMETER;
  /*为 body 的 body 字段分配空间*/
  body->body = (char *) osip_malloc (length + 1);
  if (body->body == NULL)
    return OSIP NOMEM;
    /*以内存复制的方式将 start_of_body 中的内容复制长度为 length 到地址为 body-
    >body 处*/
  memcpy (body->body, start_of_body, length);
  /*填充字符串结尾标志*/
  body->body[length] = '\0';
  /*设置 body 长度*/
  body->length = length;
  return OSIP SUCCESS;
}
```

## 20.4.4　body 类型到字符串类型转换函数 osip_body_to_str()

函数 osip_body_to_str()与函数 osip_body_parse()对应，用于将 body 中的 body 字段的内容复制到参数 dest 中。

```
Int osip_body_to_str (const osip_body_t * body, char **dest, size_t *
str_length)
{
  char *tmp_body;
  char *tmp;
  char *ptr;
  int pos;
  int i;
  size_t length;

  *dest = NULL;
  *str_length = 0;
  /*用于安全性检查部分*/
  if (body == NULL)
    return OSIP_BADPARAMETER;
  if (body->body == NULL)
    return OSIP BADPARAMETER;
  if (body->headers == NULL)
    return OSIP_BADPARAMETER;
  if (body->length <= 0)
    return OSIP_BADPARAMETER;
  /*分配空间用于保存复制中间变量*/
  length = 15 + body->length + (osip_list_size (body->headers) * 40);
  tmp_body = (char *) osip_malloc (length);
```

```
if (tmp_body == NULL)
  return OSIP_NOMEM;
ptr = tmp_body;                  /*保存字符串的初始地址*/

if (body->content type != NULL)
  {
  /*在 tmp_body 后面加上字符串 content-type:*/
    tmp_body = osip_strn_append (tmp_body, "content-type: ", 14);
    /* 将 body->content_type 转化为字符串保存在 tmp 中*/
    i = osip_content_type_to_str (body->content_type, &tmp);
    if (i != 0)
      {
        /*转换失败则释放空间并退出*/
        osip free (ptr);
        return i;
      }
    /*检查 tmp_body 的空间是否够用，不够则重新分配*/
    if (length < tmp body - ptr + strlen (tmp) + 4)
      {
        size_t len;

        len = tmp_body - ptr;
        length = length + strlen (tmp) + 4;
        ptr = osip realloc (ptr, length);
        tmp_body = ptr + len;
      }
    /*将 tmp 添加到 tmp_body 后面*/
    tmp_body = osip_str_append (tmp_body, tmp);
    /*释放为 tmp 分配的空间*/
    osip_free (tmp);
    /*在 tmp_body 后添加字符串 CRLF*/
    tmp body = osip strn append (tmp body, CRLF, 2);
  }

pos = 0;
while (!osip list eol (body->headers, pos))
  {
    osip_header_t *header;
    /*获得 body 头部*/
    header = (osip_header_t *) osip_list_get (body->headers, pos);
    /*将得到的头部转化为字符串保存在 tmp 中*/
    i = osip_header_to_str (header, &tmp);
    if (i != 0)
      {
        /*转换失败则释放空间并退出*/
        osip free (ptr);
        return i;
      }
    /*检查 tmp_body 的空间是否够用，不够则重新分配*/
    if (length < tmp body - ptr + strlen (tmp) + 4)
      {
        size_t len;

        len = tmp body - ptr;
        length = length + strlen (tmp) + 4;
        ptr = osip realloc (ptr, length);
        tmp body = ptr + len;
      }
```

```
      /*将 tmp 添加到 tmp_body 后面*/
      tmp_body = osip_str_append (tmp_body, tmp);
      /*释放为 tmp 分配的空间*/
      osip_free (tmp);
      /*在 tmp_body 后添加字符串 CRLF*/
      tmp_body = osip_strn_append (tmp_body, CRLF, 2);
      pos++;
    }

  if ((osip_list_size (body->headers) > 0) || (body->content_type != NULL))
    {
      tmp_body = osip_strn_append (tmp_body, CRLF, 2);
    }
  if (length < tmp_body - ptr + body->length + 4)
    {
      size_t len;

      len = tmp_body - ptr;
      length = length + body->length + 4;
      ptr = osip_realloc (ptr, length);
      tmp_body = ptr + len;
     }
    /*将 body 上的 body 字段向地址为 tmp_body 处复制长度为 body->length 个字符*/
  memcpy (tmp_body, body->body, body->length);
  tmp_body = tmp_body + body->length;

  /* end of this body */
  if (str_length != NULL)
    *str_length = tmp_body - ptr;
  *dest = ptr;
  return OSIP_SUCCESS;
}
```

## 20.4.5　克隆函数 osip_body_clone()

函数 osip_body_clone()将 body 复制给 dest。

```
Int osip_body_clone (const osip_body_t * body, osip_body_t ** dest)
{
  int i;
  osip_body_t *copy;
  /*安全性检查*/
  if (body == NULL || body->length <= 0)
    return OSIP_BADPARAMETER;
  /*创建一个新的 body 类型*/
  i = osip_body_init (&copy);
  if (i != 0)
    return i;
  /*为新建的类型的 body 字段分配空间*/
  copy->body = (char *) osip_malloc (body->length + 2);
  /*分配失败则退出*/
  if (copy->body==NULL)
  return OSIP_NOMEM;
  /*设置新建结构体 copy 的长度*/
  copy->length = body->length;
  /*复制 body 的字段 body 到 copy 的字段 body*/
```

```
  memcpy (copy->body, body->body, body->length);
  /*设置字符结束标志*/
  copy->body[body->length] = '\0';

  if (body->content_type != NULL)
{
  /*克隆 content_type*/
      i = osip_content_type_clone (body->content_type, &(copy->content_
      type));
      if (i != 0)
  {
       /*克隆 content_type 失败则退出且释放空间*/
       osip body free (copy);
       return i;
  }
    }
  /*克隆结构体的头部*/
  i = osip list clone(body->headers, copy->headers, &osip header clone);
  if (i != 0)
  {
  /*克隆结构体的头部失败则退出*/
  osip body free (copy);
  return i;
  }
  /*将新创建并进行了克隆的结构体赋给 dest 指针*/
  *dest = copy;
  return OSIP SUCCESS;
}
```

## 20.4.6　oSIP 解析器分类

oSIP 解析器分为 3 类：SIP 解析器、SDP 解析器和 URL 解析器。前面提到的是 SIP 解析器，另外还有两类解析器。3 类解析器的作用分别如下：

- SIP 解析器主要用于解析 SIP 头域及其相应的操作。
- SDP 解析器用于解析 SDP 包及其相关的操作。
- URL 解析器用于处理 SIP URI 的 host、port、username、password 和 scheme 等 get()和 set()操作。

SIP URI 是通过 SIP 呼叫对方的 SIP 地址方案，即一个 SIP URI 就是一个用户的 SIP 电话号码。SIP URI 类似电子邮件地址，书写格式如下：

```
SIP URI = sip:a@b:Port
```

其中，a 表示用户名，b 表示服务主机（域名或 IP）。下面举几个例子进行说明 SIP URI 的书写方法。例子如下：

```
sip:bob@212.123.1.213          //bob 为用户名，212.123.1.213 为服务器 IP 地址
sip:bob@biloxi.com             //bob 为用户名，biloxi.com 为服务器主机名
sip:5201314@biloxi.com         //5201314 为用户名，biloxi.com 为服务器主机名
```

# 20.5　oSIP 事务层

SIP 是一个基于事务处理的协议：部件之间的交互是通过一系列的消息交换所完成的。特别是，一个 SIP 事务由一个单个请求和这个请求的所有应答组成，这些应答包括了零个或者多个临时应答以及一个或者多个终结应答。

事务分为客户端和服务端两方。客户端的事务是客户端事务，服务器端的事务就是服务端事务。客户端事务发出请求，并且服务端事务送回应答。事务层比较重要的两个概念就是事务和事件，在 oSIP 中用结构体 osip_transaction_t 和结构体 osip_event_t 表示。

结构体 osip_transaction_t 的定义：

```
typedef struct osip_transaction osip_transaction_t;
struct osip transaction
  {
    void *your_instance;              /**用户定义指针*/
    int transactionid;                /**内部事务 ID*/
    osip_fifo_t *transactionff;       /**用于存放事件的 fifo 队列*/

    osip_via_t *topvia;               /**< CALL-LEG definition (Top Via)*/
    osip_from_t *from;                /**< CALL-LEG definition (From)*/
    osip_to_t *to;                    /**< CALL-LEG definition (To)*/
    osip_call_id_t *callid;           /**< CALL-LEG definition (Call-ID)*/
    osip_cseq_t *cseq;                /**< CALL-LEG definition (CSeq)*/

    osip_message_t *orig_request;     /**初始请求*/
    osip_message_t *last_response;    /**最后响应*/
    osip_message_t *ack;              /**应答请求*/

    state_t state;                    /**事务的当前状态*/

    time_t birth_time;                /**事务开始时间*/
    time_t completed_time;            /**事务结束时间*/

    int in_socket;        /**< Optional socket for incoming message*/
    int out_socket;       /**< Optional place for outgoing message*/

    void *config;         /**@internal transaction is managed by osip_t*/

    osip_fsm_type_t ctx_type;    /**事务类型*/
    osip_ict_t *ict_context;     /**INVITE CLIENT TRANSACTION 结构体*/
    osip_ist_t *ist_context;     /**NON-INVITE CLIENT TRANSACTION 结构体*/
    osip_nict_t *nict_context;   /**INVITE SERVER TRANSACTION 结构体*/
    osip_nist_t *nist_context;   /**NON-INVITE SERVER TRANSACTION 结构体*/

osip_srv_record_t record;   /**记录 SERVER 入口信息的结构体*/
  };
```

结构体 osip_event_t 的定义：

```
typedef struct osip event osip event t;
struct osip_event
```

```
{
  type_t type;                 /**事件类型*/
  int transactionid;       /**与 osip 事务相关的 ID*/
  osip_message_t *sip;     /**< SIP message (optional)*/
};
```

处理事务的函数在 osip_tarnsaction.c 中定义，函数 osip_transaction_init()用于事务的初始化，函数 osip_transaction_free()用于从 osip 栈中移除事务，函数 osip_transaction_add_event()用于向事务的 fifo 队列中添加事件，函数 osip_transaction_execute()用于执行事务处理，调用前面分析过的状态机处理事务过程。由于篇幅的限制这里只分析事务初始化函数 osip_transaction_init()，其他函数的定义读者参考源代码自行分析其实现细节。

事务初始化函数 osip_transaction_init()用于构造一个事务，并初始化该事务的各个字段，构造该事务的 fifo 队列用于存放事件，并初始化该事件队列，根据输入事务类型设置其事务的类型，根据事务的类型设置事务的初始状态，并初始化此状态机。省略函数 osip_transaction_init()的安全检查、trace 机制等，其核心部分代码如下：

```
int osip_transaction_init (osip_transaction_t ** transaction,
                   osip_fsm_type_t ctx_type, osip_t * osip,
                   osip_message_t * request)
{
  static int transactionid = 1;
  osip_via_t *topvia;

  int i;
  time_t now;
  /*为构造的事务结构体分配空间*/
  *transaction = (osip_transaction_t *) osip_malloc (sizeof (osip_
  transaction_t));
  /*获得当前时间*/
  now = time (NULL);
  /*初始化事务结构体*/
  memset (*transaction, 0, sizeof (osip_transaction_t));
  /*设置事务的创建时间和 ID，事务计数自动加 1*/
  (*transaction)->birth_time = now;
  (*transaction)->transactionid = transactionid;
  transactionid++;
  /*从输入参数 request 中获得字段 vias 的值赋给 topvia*/
  topvia = osip list get (&request->vias, 0);
  /*设置事务的 topvia 字段*/
  __osip_transaction_set_topvia (*transaction, topvia);
  /*下面分别对事务的 from、to、call_id、cseq 字段进行设置*/
  i = __osip_transaction_set_from (*transaction, request->from);
  i = __osip_transaction_set_to (*transaction, request->to);
  i = __osip_transaction_set_call_id (*transaction, request->call_id);
  i = __osip_transaction_set_cseq (*transaction, request->cseq);
  /*设置事务的 orig_request 和 config*/
  (*transaction)->orig_request = NULL;
  (*transaction)->config = osip;
  /*为事务的事件队列分配空间*/
  (*transaction)->transactionff = (osip_fifo_t *) osip_malloc (sizeof
  (osip_fifo_t));
  /*初始化事件队列*/
  osip_fifo_init ((*transaction)->transactionff);
```

```
  /*设置事务类型*/
  (*transaction)->ctx_type = ctx_type;
  /*将事务的 ict_context 、ist_context 、nict_context 和 nist_context 均设置
  为空 */
  (*transaction)->ict_context = NULL;
  (*transaction)->ist_context = NULL;
  (*transaction)->nict_context = NULL;
  (*transaction)->nist_context = NULL;
  /*如果事务的类型为 ICT*/
  if (ctx type == ICT)
{
      /*设置事务的初始状态，并初始化 ict 状态机*/
      (*transaction)->state = ICT_PRE_CALLING;
      __osip_ict_init (&((*transaction)->ict_context), osip, request);
      /*将事务加入 ict 事务处理队列*/
      __osip_add_ict (osip, *transaction);
  } else if (ctx_type == IST)
{
      /*设置事务的初始状态，并初始化 ist 状态机*/
      (*transaction)->state = IST_PRE_PROCEEDING;
      i = __osip_ist_init (&((*transaction)->ist_context), osip, request);
      /*将事务加入 ist 事务处理队列*/
      __osip_add_ist (osip, *transaction);
  } else if (ctx_type == NICT)
{
      /*设置事务的初始状态，并初始化 nict 状态机*/
      (*transaction)->state = NICT_PRE_TRYING;
      i = __osip_nict_init (&((*transaction)->nict_context), osip,
      request);
      /*将事务加入 nict 事务处理队列*/
      __osip_add_nict (osip, *transaction);
  } else
{
      /*设置事务的初始状态，并初始化 nist 状态机*/
      (*transaction)->state = NIST_PRE_TRYING;
      i = __osip_nist_init (&((*transaction)->nist_context), osip,
      request);
      /*将事务加入 nist 事务处理队列*/
      __osip_add_nist (osip, *transaction);
    }
  return OSIP_SUCCESS;
}
```

## 20.6　SIP 建立会话的过程

下面采用序列图的方式表示 A、B 两个用户间通过 SIP 消息交换建立会话的过程，如图 20.12 所示。A 通过 B 的 SIP 标志“呼叫”B，这个 SIP 标志是统一分配的资源（Uniform Resource Identifier URI）称作 SIP URI。它很像一个 E-mail 地址，典型的 SIP URI 包括一个用户名和一个主机名。在这个范例中，SIP URI 是 sip:bbb@bwww.com，bwww.com 是 B 的 SIP 服务提供商。A 有一个 SIP URI: sip:aaa@awww.com。

下面是 A 通过自己的 Softphone 呼叫 B 的 SIP phone，并建立和 B 的会话的整个过程。A 不知道 B 或者 B 的 SIP 服务器的位置，所以 A 首先将请求发送到 A 的 SIP 服务器 awww.com。SIP 服务器 awww.com 收到 INVITE 请求后，回应 100（Trying）给 A 的 Softphone，并在 via 头上加入自己的地址转发 INVITE 请求给 B 的 SIP 服务器 bwww.com。B 的 SIP 服务器 bwww.com 收到该 INVITE 请求后回应 100（Trying）给代理服务器 awww.com，并在 via 头上加入自己的地址转发 INVITE 请求给 B 的 SIP phone。到此时 B 的 SIP phone 就提示 B，A 在呼叫他。B 的 SIP phone 这时会发送一个 180（ringing）响应，该回应会通过原路返回给 A。当 B 接通 SIP phone 时，SIP phone 就会发送 200（OK）回应，该回应最后到达 A 的 Softphone，到此 A 和 B 就可以进行语音或视频通话。通话结束后，B 挂机，B 的 SIP phone 就会发送 BYE 给 A 的 Softphone，A 的 Softphone 会回应 200（OK）。

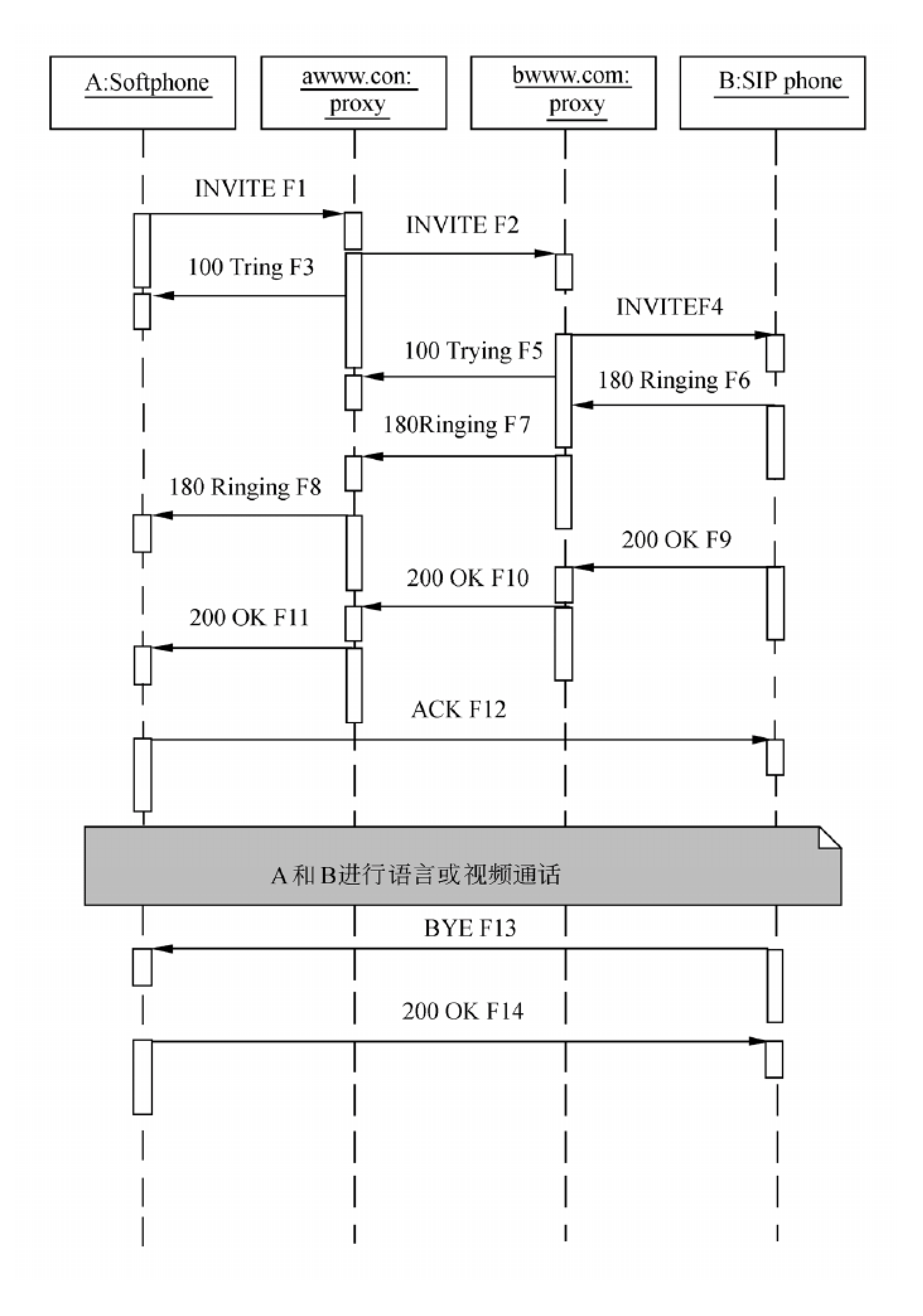



图 20.12　SIP 会话建立过程

# 20.7　RTP 协议

RTP（real-time transport protocol）实时传输协议，在多点传送（多播）或单点传送（单播）的网络服务上，提供端到端的网络传输功能，适合应用程序传输实时数据，如音频、视频或者仿真数据。通过 SIP 协议建立起会话，就可以通过 RTP 传输这些会话的实时数据。

介绍 RTP 协议时，首先了解一下 RTP 协议的一些基本概念，然后针对代码看 RTP 是如何发送和接收 RTP 包，然后细化这个过程。

## 20.7.1　RTP 基本概念

RTP 提供实时的端到端的数据传输服务。传输的数据类型有交互式的音频和视频。服务的内容包括有效载荷、序列号、时间戳和传输监测控制。应用程序在 UDP 上运行 RTP 来使用它的多路技术与校验和服务。RTP 本身不提供任何机制确保实时传输或服务质量保证，而是由低层的服务来保证。它不保证下层网络可靠，也不保证按顺序传送数据包。RTP 包含的序列号使得接受方可以重构发送方的数据包顺序，但序列号也可以确定一个数据包的正确位置，例如，在视频解码时不用按顺序地对数据包进行解码。

RTP 包括两个密切相关的部分：

- 实时传输协议 RTP，主要用于实时数据传输。
- RTP 控制协议 RTCP，用于服务质量监控与反馈、媒体间的同步、传达会议中参与者的信息。不必支持一个应用程序中所有的通信控制条件。

在分析源码前需要了解相关的基本概念，下面列出了一些 RTP 的基本概念，对这些概念深入的了解可以参看 RFC3550。

- RTP 负载（RTP payload）：RTP 包中的净数据，例如，音频样本或压缩好的视频数据。
- RTP 包（RTP packet）：一种数据包，其组成部分包括 RTP 包固定的报头、可能为空的作用源（contributing sources）列表和负载数据。下层协议对 RTP 包的进行封包时，一个包中可以包含一 RTP 包，也可包含多个 RTP 包。
- RTCP 包（RTCP packet）：一种控制包，其组成部分包括 RTCP 包固定的报头和一个结构化的部分，该部分具体元素还应该根据不同 RTCP 包的类型而定。一般下层协议的包由多个 RTCP 包组成。
- 端口（Port）：用于在同一主机中区分不同目的地。RTP 需要依靠低层协议提供的多种机制，如“端口”用于多路复用会话中的 RTP 包和 RTCP 包。
- 传输地址（Transport address）：是网络地址与端口的结合，用来指定一个唯一的传输层次的终端，例如，一个 IP 地址与一个 UDP 端口的组合。包是从源传输地址发往目的传输地址。
- RTP 媒体类型（RTP media type）：一个 RTP 媒体类型是一个单独 RTP 会话所承载的负载类型的集合。RTP 配置文件中将 RTP 媒体类型分配给 RTP 负载类型。

- 多媒体会话（Multimedia session）：在一个会话公共组中，并发的 RTP 会话的集合。例如，一个视频会议（为多媒体会话）中，可能包含一个音频 RTP 会话和一个视频 RTP 会话。
- RTP 会话（RTP session）：一组参与者通过 RTP 协议进行通信时所产生的关联。一个参与者可能同时参与多个 RTP 会话。在一个多媒体会话中，除了编码方式将多种媒体多路复用到单一数据流中外，每种媒体都将使用各自的 RTCP 包，通过单独的 RTP 会话进行传送。通过使用不同的目的传输地址对（一个网络地址加上一对分别用于 RTP 和 RTCP 的端口，构成了一个传输地址对）来接收不同的会话，参与者能将多个 RTP 会话分隔开。IP 多播时，单个 RTP 会话中的全部参与者，共享一个公用目的传输地址对；单播时，参与者将使用不同的目的传输地址对，个体单播网络地址加上一个端口对。对于单播而言，参与者可以使用相同的端口对收听其他全部参与者，也可能使用不同的端口对分别收听其他各个参与者。

## 20.7.2　发送 RTP

RTP 的发送过程在 rtpsend.c 中定义，该文件的 main()函数清晰地描述构造 RTP、设置 RTP Session 到发送 RTP 包的整个过程，下面是其代码：

```
int main(int argc, char *argv[])
{
RtpSession *session;
unsigned char buffer[160];
int i;
FILE *infile;
char *ssrc;
uint32_t user_ts=0;
int clockslide=0;
int jitter=0;
if (argc<4){
    printf("%s", help);
    return -1;
}
for(i=4;i<argc;i++){
    if (strcmp(argv[i],"--with-clockslide")==0){
        i++;
        if (i>=argc) {
            printf("%s", help);
            return -1;
        }
        /*将输入的参数字符串转化为整数类型*/
        clockslide=atoi(argv[i]);
        ortp message("Using clockslide of %i milisecond every 50
        packets.",clockslide);
    }else if (strcmp(argv[i],"--with-jitter")==0){
        ortp_message("Jitter will be added to outgoing stream.");
        i++;
        if (i>=argc) {
            printf("%s", help);
            return -1;
        }
        jitter=atoi(argv[i]);
```

```
    }
    }
    /*RTP 库的初始化，在调用 RTP 相关的 API 之前首先调用该函数*/
    ortp_init();
    /*初始化 RTP 调度程序*/
    ortp_scheduler_init();
    /*设置日志文件的级别，该函数一般用于开发或者调试中追踪相关的信息*/
    ortp_set_log_level_mask(ORTP_MESSAGE|ORTP_WARNING|ORTP_ERROR);
    /*创建一个 RTP  session，用于发送数据*/
    session=rtp_session_new(RTP_SESSION_SENDONLY);
    /*设置 session 的调度模式*/
    rtp_session_set_scheduling_mode(session,1);
    /*设置 session 为阻塞模式，该模式默认在调度程序中执行，因此设置该模式时同时会设置
    Session 为调度模式*/
    rtp_session_set_blocking_mode(session,1);
    /*设置连接模式，如果设置了连接模式，当一个 socket 到达目的地时将会调用系统的
    connect()函数*/
    rtp_session_set_connected_mode(session,TRUE);
    /*设置 RTP 的远程地址，即 RTP 包准备发往的地址*/
    rtp_session_set_remote_addr(session,argv[2],atoi(argv[3]));
    /*设置期望的 session 负载类型*/
    rtp_session_set_payload_type(session,0);
    /*ssrc 的作用：在随机的时间间隔中，一个参与者必须检测其他参与者是否已经超时。为此，
    对接收者（we_sent 为 false），要计算决定性时间间隔 Td，如果从时刻 Tc-M*Td（M 为
    超时因子，默认为 5 秒）开始，未发送过 RTP 或 RTCP 包，则超时。其 SSRC 将被从列表中移
    除，成员被更新。在发送者列表中也要进行类似的检测。发送者列表中，任何从时间 tc-2T
    （在最后两个 RTCP 报告时间间隔内）未发送 RTP 包的发送者，其 SSRC 从发送者列表中移除，
    列表更新*/
    ssrc=getenv("SSRC");
    if (ssrc!=NULL) {
        printf("using SSRC=%i.\n",atoi(ssrc));
        rtp_session_set_ssrc(session,atoi(ssrc));
    }
    /*针对不同平台的可移植性的系统函数，打开文件操作*/
    #ifndef  WIN32
    infile=fopen(argv[1],"r");
    #else
    infile=fopen(argv[1],"rb");
    #endif
    /*对于打开文件操作的安全性检查*/
    if (infile==NULL)  {
        perror("Cannot open file");
        return -1;
    }
    /*系统收到信号 SIGINT 就会通过传入的地址调用函数 stophandler ()，该函数就是设置
    runcond =0*/
    signal(SIGINT,stophandler);
    while( ((i=fread(buffer,1,160,infile))>0)  &&  (runcond) )
    {
        /*从缓冲区中发送带有时间戳的 RTP 数据包到目的地址*/
        rtp_session_send_with_ts(session,buffer,i,user_ts);
        user_ts+=160;
        if (clockslide!=0  && user_ts%(160*50)==0){
            ortp_message("Clock sliding of %i miliseconds now",clockslide);
            /*设置时间误差*/
            rtp_session_make_time_distorsion(session,clockslide);
```

```
        }
        /*下面是模拟的突发延迟包*/
        if (jitter && (user_ts%(8000)==0)) {
            struct timespec pausetime, remtime;
            ortp message("Simulating late packets now (%i milliseconds)",
            jitter);
            pausetime.tv_sec=jitter/1000;
            pausetime.tv_nsec=(jitter%1000)*1000000;
            while(nanosleep(&pausetime,&remtime)==-1 && errno==EINTR){
                pausetime=remtime;
            }
        }
    }
    /*关闭文件*/
    fclose(infile);
    /*关闭 Session*/
    rtp session destroy(session);
    /*退出 RTP，包括关闭调度程序*/
    ortp exit();
    /*打印 RTP 包的一些情况，包括收到包、丢失包等信息*/
    ortp global stats display();
    return 0;
}
```

## 20.7.3　接收 RTP

RTP 的接收过程在 rtprecv.c 中描述，接收过程包括初始化 RTP、初始化调度器、创建接收 Session、设置连接模式、设置对称 RTP、接收数据包存放在缓冲区、将缓冲区数据写入文件、关闭 Session、退出 RTP 等过程。下面是该接收 RTP 过程的代码：

```
    int main(int argc, char*argv[])
    {
    RtpSession *session;
    unsigned char buffer[160];
    int err;
    uint32_t ts=0;
    int stream_received=0;
    FILE *outfile;
    int local_port;
    int have more;
    int i;
    int format=0;
    int soundcard=0;
    int sound_fd=0;
    int jittcomp=40;
    bool t adapt=TRUE;
    /*将第 2 个参数字符串转化为整数类型的本地端口号*/
    local port=atoi(argv[2]);
    if (local_port<=0) {
        printf("%s",help);
        return -1;
    }
     /*根据参数指定采用哪种 PCM 编码算法，u 律还是 A 律*/
    for (i=3;i<argc;i++)
    {
        if (strcmp(argv[i],"--noadapt")==0) adapt=FALSE;
```

```
        if (strcmp(argv[i],"--format")==0){
            i++;
            if (i<argc){
                if (strcmp(argv[i],"mulaw")==0){
                    format=MULAW;
                }else
                if (strcmp(argv[i],"alaw")==0){
                    format=ALAW;
                }else{
                    printf("Unsupported format %s\n",argv[i]);
                    return -1;
                }
            }
        }
        else if (strcmp(argv[i],"--soundcard")==0){
            soundcard=1;
        }
        else if (strcmp(argv[i],"--with-jitter")==0){
            i++;
            if (i<argc){
                jittcomp=atoi(argv[i]);
                printf("Using a jitter buffer of %i milliseconds.\n",
                jittcomp);
            }
        }
    }
    /*打开输出文件，准备向文件中写内容*/
    outfile=fopen(argv[1],"wb");
    if (outfile==NULL) {
        perror("Cannot open file for writing");
        return -1;
    }
     /*声卡初始化包括打开声音设备文件、设置采样频率、设置声道数、设置音频数据格式*/
    if (soundcard){
        sound_fd=sound_init(format);
    }
    /*RTP 初始化，在调用 RTP 库前必须要做的工作*/
    ortp_init();
    /*RTP 调度程序初始化*/
    ortp_scheduler_init();
    /*设置日志文件的级别，该函数一般用于开发或者调试中追踪相关的信息*/
    ortp_set_log_level_mask(ORTP_DEBUG|ORTP_MESSAGE|ORTP_WARNING|
    ORTP_ER ROR);
    /*系统收到信号 SIGINT 就会通过传入的地址调用函数 stop_handler ()，该函数就是设
    置 cond =0*/
    signal(SIGINT,stop_handler);
    /*创建一个新的 Session，用于接收 RTP 数据包*/
    session=rtp_session_new(RTP_SESSION_RECVONLY);
    /*设置为调度模式*/
    rtp_session_set_scheduling_mode(session,1);
    /*设置 Session 为阻塞模式，该模式默认在调度程序中执行，因此设置该模式时同时会设置
    session 为调度模式，在该函数中调用了函数 rtp_session_set_scheduling_mode
    (session,1);*/
    rtp_session_set_blocking_mode(session,1);
    /*这里设置本地地址用于监听 RTP 包，如果本地地址没有设置，则采用默认的 IP 地址和任意
    端口*/
    rtp_session_set_local_addr(session,"0.0.0.0",atoi(argv[2]));
    /*设置连接模式，如果设置了连接模式，当一个 socket 到达目的地时将会调用系统的
```

```
    connect()函数*/
    rtp session set connected mode(session,TRUE);
    /*设置为对称 RTP*/
    rtp session set symmetric rtp(session,TRUE);
    /*设置可以调整时延或者抖动*/
    rtp session enable adaptive jitter compensation(session,adapt);
    /*设置时延或者抖动的补偿参数*/
    rtp session set jitter compensation(session,jittcomp);
    /*设置载荷类型*/
    rtp_session_set_payload_type(session,0);
    /*当用户函数支持的信号量在处理的过程中发生了变化时，用户注册的回调函数将会被通知
    */
    rtp session signal connect(session,"ssrc changed",(RtpCallback)
    ssrc    cb,0);
    rtp_session_signal_connect(session,"ssrc_changed",(RtpCallback)
    rtp s   ession reset,0);

    while(cond)
    {
        have_more=1;
        while (have more){
            /*接收到来的 RTP 流存放到缓冲区*/
            err=rtp session recv with ts(session,buffer,160,ts,&have
            more);
            if (err>0) stream received=1;
            /*在第一个 RTP 数据包返回前防止写静音数据*/
            if ((stream received) && (err>0)) {
                size_t ret = fwrite(buffer,1,err,outfile);
                if (sound_fd>0)
                    /*播放声音*/
                    ret = write(sound fd,buffer,err);
            }
        }
        ts+=160;
        //ortp_message("Receiving packet.");
    }
    /*通信结束后销毁 Session*/
    rtp session destroy(session);
    /*退出 RTP*/
    ortp exit();
    /*打印 RFP 的信息*/
    ortp global stats display();

    return 0;
}
```

## 20.8　Linphone 编译与测试

在对 Linphone 有一定了解的基础上，讲解其编译和测试过程。其编译包括 X86 平台的编译和 ARM 平台的编译。

## 20.8.1　编译 Linphone 需要的软件包

编译 Linphone 依赖一些相关的库，这里列出编译这些库的软件包，同时要注意采用的版本。下面是笔者编译过程中使用的软件包和对应的版本。

linphone-3.2.1.tar.gz 依赖的库如下：

- libogg-1.1.3.tar.gz；
- speex-1.2beta3.tar.gz（依赖于 libogg）；
- libosip2-3.3.0.tar.gz；
- libeXosip2-3.1.0.tar.gz；
- SDL-1.2.14.tar.gz；
- ffmpeg-0.5.1.tar.gz（依赖 SDL）。

在编译前读者可以查看 linphone 目录下的 README 文件，其中介绍了版本依赖关系，下面为 linphone-3.2.1 所依赖的相关软件包的信息：

```
*****************Building linphone **********************************
- you need at least:
              - libosip2>=3.0.3
              - libeXosip2>=3.0.3
              - speex>=1.1.6
              - libreadline
       + gsm codec (gsm source package or libgsm-dev or gsm-devel) (optional)
       + if you want to gtk/glade interface:
              - gtk>=2.16.0
              - libglade>=2.2
       + if you want video support:
              - SDL>=1.2.10
              - libavcodec (ffmpeg) from a year 2007 or later cvs/svn
              - libswscale (part of ffmpeg too) for better scaling
              performance
```

## 20.8.2　X86 平台上编译和安装

编译和安装有一定的依赖关系，下面将依次介绍具体每个软件包的编译和安装过程，同时列出编译和安装过程中遇到的问题和相应的解决方法。

### 1．建立相关目录

在/usr/local 目录下建立本次编译目录 linphone，然后在该目录下建立存放源码目录 src，建立 X86 的安装目录 linphone_x86。复制下载的 20.4.1 列出的源码放在 src 目录下。

### 2．编译libogg

libogg 提供了对多媒体格式文件操作的接口，是编译 speex 所依赖的文件，下面给出其集体编译和安装过程：

```
# tar zxvf libogg-1.1.3.tar.gz
# cd libogg-1.1.3
```

```
# ./configure --prefix=/usr/local/linphone/linphone_x86    //指定其安装目录
# make
# make install
```

### 3．编译speex

speex 不是为移动电话的语音歌曲格式编码设计的，而是专门为网络包和 VOIP 设计的。下面给出其编译和安装过程：

```
# tar zxvf speex-1.2beta3.tar.gz
# cd speex-1.2beta3
# ./configure --prefix=/usr/local/linphone/linphone_x86  //指定其安装目录
# make
# make install
```

### 4．编译libosip2

libosip2 是 SIP 协议的实现库，它的目的是提供给多媒体和电信软件设计者一个方便和强大的接口，让他们在这个接口基础上开发自己的 SIP 应用。下面是其详细的编译和安装过程：

```
# tar zxvf libosip2-3.3.0.tar.gz
# cd libosip2-3.3.0
# ./configure --prefix=/usr/local/linphone/linphone_x86
# make
# make install
```

### 5．编译libeXosip

libeXosip2 是 eXosip 的库文件，eXosip 是协议栈 Osip2 的扩展协议集，它部分封装了协议栈 Osip2，使得它更容易被使用。下面是编译安装 eXosip 库文件的过程：

```
# tar zxvf libeXosip2-3.1.0.tar.gz
# cd libeXosip2-3.1.0
# ./configure --prefix=/usr/local/linphone/linphone x86 \
PKG_CONFIG_PATH=/usr/local/linphone/linphone_x86/lib/pkgconfig
//pkg-config 的有效路径
# make
# make install
```

**注意**：如果上一步的 libosip2 不是按照默认路径安装，那么在安装 libeXosip 时要在 configure 后面添加 PKG_CONFIG_PATH=安装目录\lib\pkgconfig，如果是默认方式安装则不需要此参数。

### 6．编译SDL

SDL（Simple DirectMedia Layer）是一个跨平台的多媒体库，是用于直接控制底层多媒体硬件的接口。这些多媒体功能包括了音频、键盘和鼠标（事件）、游戏摇杆等。编译 ffmpeg 时需要依赖该库，下面给出其编译安装过程：

```
# tar zxvf SDL-1.2.14.tar.gz
# cd SDL-1.2.14
# ./configure --prefix=/usr/local/linphone/linphone x86
# make
# make install
```

### 7．编译ffmpeg

ffmpeg 用于视频文件转换，支持通过实时抓取电视卡并编码成视频文件。在编译 mediastreamer2 时需要该库的支持。下面是其编译和安装过程：

```
# tar zxvf ffmpeg-0.5.1.tar.gz
# cd ffmpeg-0.5.1
```

在使用 configure 时，先查看下其参数的使用，笔者在使用 ffmpeg-0.4.9.tar.gz 安装包时，发现两个版本的安装文件 configure 所带的参数相差很多。

```
# ./configure -help
#  ./configure  --prefix=/usr/local/linphone/linphone_x86  --enable-gpl
--enable-shared  \
 --enable-swscale --enable-pthreads
```

使用 configure 命令生成 Makefile 后，修改 Makefile 手动添加 X11 库，添加方法如下：

```
OBJS = ffmpeg.o ffserver.o cmdutils.o $(FFPLAY_O)
SRCS = $(OBJS:.o=.c) $(ASM OBJS:.o=.s)
FFLIBS = -L./libavformat -lavformat$(BUILDSUF) -L./libavcodec -lavcodec$
(BUILDSUF) -L./libavutil -lavutil$(BUILDSUF)
```

在其后添加 X11 库支持，修改后为：

```
OBJS = ffmpeg.o ffserver.o cmdutils.o $(FFPLAY O)
SRCS = $(OBJS:.o=.c) $(ASM_OBJS:.o=.s)
FFLIBS = -L./libavformat -lavformat$(BUILDSUF) -L./libavcodec -lavcodec$
(BUILDSUF) -L./libavutil -lavutil$(BUILDSUF) -lX11
```

修改完成后保存，使用 make 和 make install 进行安装和编译。

```
# make
# make install
```

### 8．编译oRTP

oRTP 是实时传输协议栈，其代码就包含在 linphone 目录下，下面给出其编译和安装过程：

```
# tar zxvf linphone-3.2.1.tar.gz
# cd linphone-3.2.1
# cd oRTP
# ./configure --prefix=/usr/local/linphone/linphone_x86
# make
# make install
```

### 9．编译mediastreamer2

mediastreamer2 是一个功能强大、轻量级流媒体引擎，专门为语音/视频电话应用而设

计，其源码也在 linphone 目录下，下面给出其编译和安装过程：

在编译安装前先增加两个宏定义，在后面的编译中会提示说没有对该宏进行定义。在/usr/local/linphone/linphone_x86/include/speex 下找到 speex_preprocess.h 文件，向其添加下面两个宏的定义。

```
#define SPEEX_PREPROCESS_GET_PSD_SIZE      34
#define SPEEX_PREPROCESS_GET_PSD          35
```

修改完成后，保存并退出，进行下面的编译安装过程：

```
# cd ../mediastreamer2
# ./configure --prefix=/usr/local/linphone/linphone_x86   \
PKG_CONFIG_PATH=/usr/local/linphone/linphone_x86/lib/pkgconfig
# make
# make install
```

### 10．编译Linphone

使用 Linphone 可以免费在互联网上与其他人进行通信，通信方式包括语音、视频、直接文本消息。下面给出其编译和安装过程：

```
# cd ..
# ./configure --prefix=/usr/local/linphone/linphone_x86   \
PKG_CONFIG_PATH=/usr/local/linphone/linphone_x86/lib/pkgconfig \
SPEEX_CFLAGS=/usr/local/linphone/linphone_x86/include/speex
--enable-gtk_ui=no
```

执行 configure 命令，有个错误: error: The intltool scripts were not found. Please install intltool.

需要安装工具 intltool。下面是其安装过程：

- 挂载系统安装光驱文件。

```
# mount -t iso9660 /dev/cdrom /mnt/cdrom/
```

- 使用“添加/删除软件”菜单安装 intltool。在“列表”选项卡中选择安装 Intltool-0.35.0-2.i386，安装该软件的时候会将其依赖的软件包一并安装，如图 20.13 所示。
- 卸载安装光驱文件。

```
# umount /mnt/cdrom/
```

安装完 Intltool 后，重新执行 configure 命令，然后执行 make 和 make install 进行编译和安装。

```
# make
# make install
```

## 20.8.3　Linphone 测试

测试的方法是通过虚拟机 Linux 和主机 Windows 进行 VoIP 测试。前面在 Linux 下面已经编译安装了 Linphone 的工具，在安装目录/usr/local/linphone/linphone_x86/bin 下。该目

录下的工具包括：

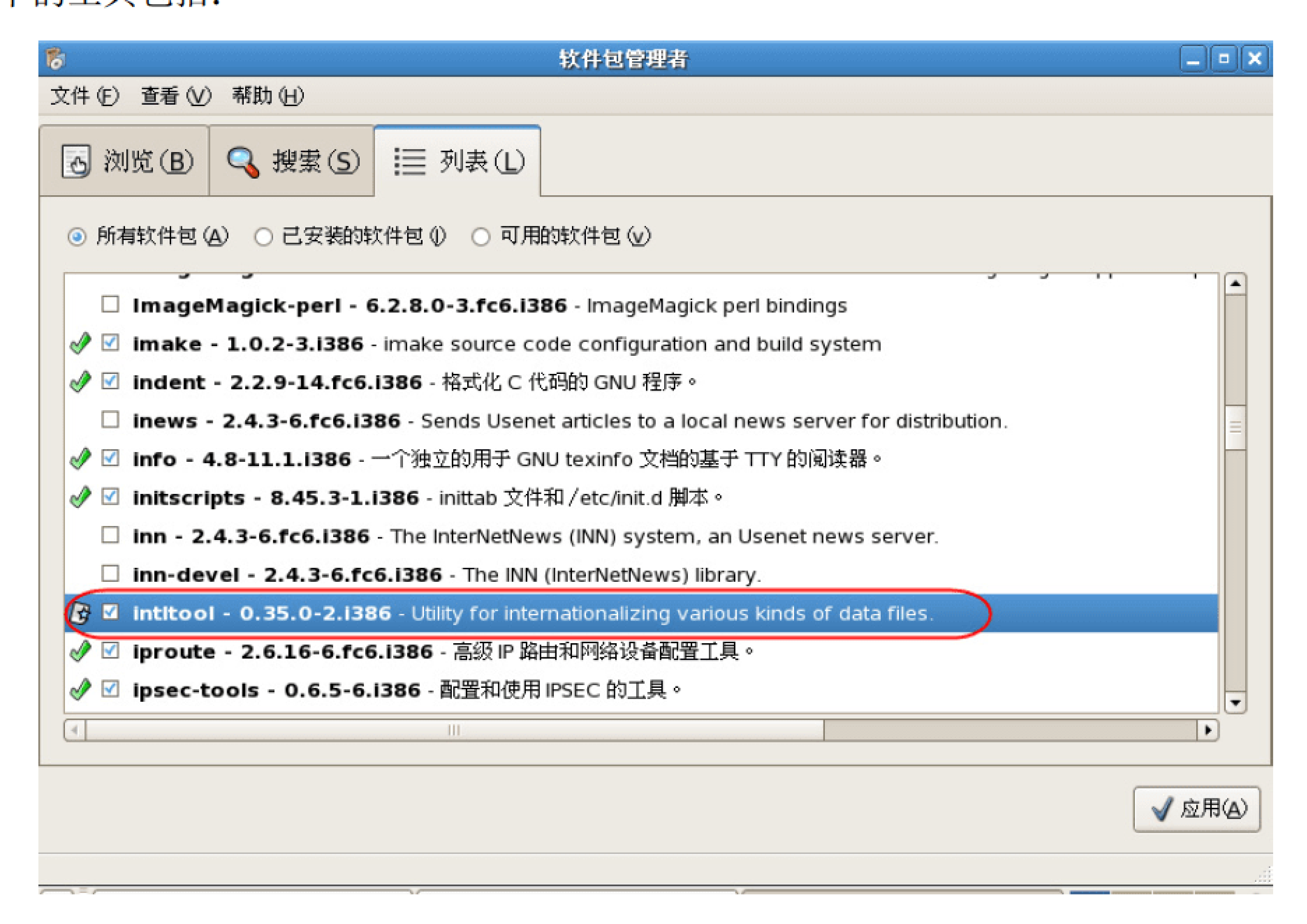



图 20.13　安装 Intltool

```
ffmpeg  ffserver   linphonecsh  sipomatic  speexdec
ffplay  linphonec  sdl-config   sip_reg    speexenc
```

### 1．Linux中运行linphonec

Linux 下运行 linphonec，使用命令为：

```
# ./linphonec -V      //参数-V 表示支持视频
```

Linux 下启动 linphonec 后，运行情况如图 20.14 所示。

### 2．Windows也安装VoIP工具

下载 linphone-3.2.1-setup.exe，在 Windows 下安装，安装完成后运行该程序，运行的结果如图 20.15 所示。可以通过 Linphone|Prefernces 对各种参数进行设置，默认情况下不需要设置。使用该工具时，下面 My current identity 处显示的是本地的 SIP 地址，上面 SIP address 输入对方的 SIP 地址，这里输入的是虚拟机的 SIP 地址：sip:root@192.168.1.123，然后鼠标单击绿色拨号按钮，就可以进行拨号了。

### 3．在Linux中进行接听

在虚拟机（Linux 系统）中会听到振铃音，同时在 Windows 中听到回铃音，Linux 中接听 VoIP 呼叫使用的命令为 answer。接通过程显示的信息为：

```
linphonec> <sip:toto@192.168.1.199:5065> is contacting you.
```

```
                                        //听到振铃音
linphonec> answer                          //输入 answer 进行接听
Connected.
linphonec> linphonec>
```

图 20.14　运行 linphonec




图 20.15　Windows 下运行的 Linphone

### 4．双方进入了通话状态

Windows 中的 Linphone 工具标识了对方的 SIP 地址和通话时长，如图 20.16 所示，视频窗口指出视频接收的对端的地址，如图 20.17 所示。

另外也可以通过 Linux 下的 linphonec 发起呼叫，在 Linux 下呼叫时，因为在同一台计算机中进行测试，如果直接使用 call sip:toto@192.168.1.199 无法正常呼叫时，可以带上端口号，带上 Windows 下 Linphone 工具指定的端口号，如果是默认的端口号 5060 则不需要

指定，收到呼叫如图 20.18 所示。

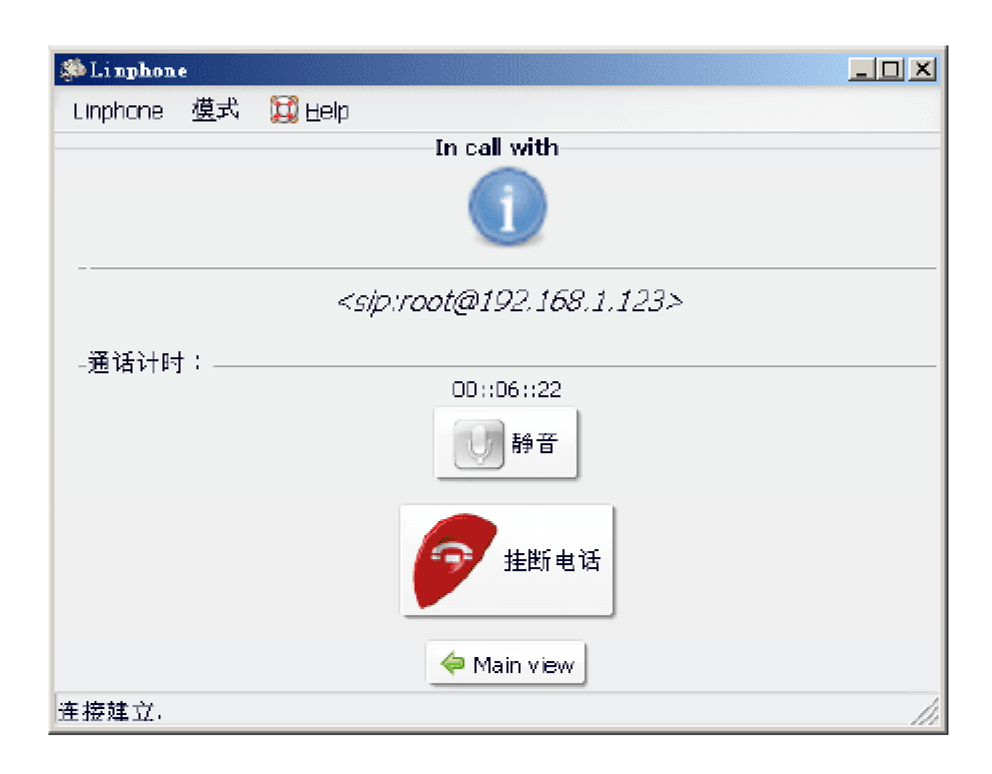



图 20.16　Linphone 主窗口




图 20.17　Linphone 视频窗口

```
# call sip:toto@192.168.1.199          //Windows 使用默认的端口号 5060 时
# call sip:toto@192.168.1.199:5066     //Windows 使用非默认的端口号 5066 时
```

呼入
来自 <sip:root@192.168.1.123> 的呼叫
接受　拒绝

图 20.18　Windows 收到 Linux 的 linphonec 呼叫

进入 linphonec 命令状态后，可以使用 help 查看 linphonec 支持的所有命令。退出通话可以使用 quit 命令或者直接使用 Ctrl+C 键退出，下面的命令读者可以一一进行测试。

```
linphonec> help
Commands are:
---------------------------
      help      Print commands help
      call      Call a SIP uri
 terminate      Terminate the current call
    answer      Answer a call
autoanswer      Show/set auto-answer mode
     proxy      Manage proxies
 soundcard      Manage soundcards
      ipv6      Use IPV6
     refer      Refer the current call to the specified destination.
       nat      Set nat address
      stun      Set stun server address
  firewall      Set firewall policy
 call-logs      Calls history
    friend      Manage friends
      play      play from a wav file
    record      record to a wav file
      quit      Exit linphonec
  register      Register in one line to a proxy
unregister      Unregister from default proxy
  duration      Print duration in seconds of the last call.
    status      Print various status information
     ports      Network ports configuration
     speak      Speak a sentence using espeak TTS engine
```

### 20.8.4　进一步的测试和开发

笔者目前测试的场景有限，读者可以继续完成下面的测试场景：

- 局域网内两台不同的主机进行音频和视频通话；
- 局域网内多台不同的主机进行音频和视频通话；
- 互联网内两台不同主机进行音频和视频通话。

笔者编译的时候没有选择 GUI 的支持，读者可以进一步编译出支持 GUI 的 linphonec 工具。

## 20.9　Linphone 交叉编译

在针对 X86 平台编译时，遇到的问题包括代码中存在的语法问题，及相关版本依赖问题，编译时选择的参数问题全部解决。而且在针对 X86 平台编译的时候，也没有采用默认的方式编译，目的是为了方便移植，那么针对 ARM 的交叉编译就非常容易了，只需要修改一条编译命令即可。必须注意的是，因为在针对 ARM 编译前，已经做过针对 X86 的编译，所以在编译每个一部分时应该先使用 make clean 进行清除，否则就会因为在编译针对 ARM 工具时使用了 X86 格式的中间文件，而导致编译出错。

### 20.9.1　Linphone 的交叉编译

编译不同版本的源代码可能使用 configure 生成 Makefile 所带的参数有所不同，下面给出的是使用交叉编译器 arm-linux-3.4.1 编译 Linphone-3.2.1 的过程。如果读者使用的版本与本例使用的版本不同，那么就应该使用./configure –help 查看配置 Makefile 所需要带的参数。

查看 linphone 目录下的文件 README.arm：

```
对编译器设置的介绍
* You need the lastest arm toolchain from http://www.handhelds.org.
Uncompress it in / .
        It contains all the cross-compilation tools. Be sure that the
        arm-linux-gcc binaries
        are in your PATH (export PATH=$PATH:/usr/local/arm/3.4.1/bin/ , for
        example)
对源码版本要求的介绍
* create within your home directory a arm/ directory, copy into it the fresh
        tarballs of libosip2>=2.2.x, speex>=1.1.6, linphone>=1.2.1
        readline>=5.1 and ncurses>=5.5 (readline needs ncurses)
         Uncompress all these
        tarballs.
对安装路径的变量的介绍
Very important things common to all packages being cross compiled:
*******************************************************************
* copy the ipaq-config.site in the ipkg/ directory of linphone into some safe
place,
for example: ~/ipaq-config.site .
* You need a directory that we call ARM_INSTALL_TREE that will own files in
```

```
the same way they will be installed on the target computer.
It is also used to build linphone over the arm binaries of its dependencies
(speex,osip,ncurses,readline).

For example:

export CONFIG_SITE=~/ipaq-config.site
export ARM_INSTALL_TREE=/armbuild

下面是安装的详细过程
Cross compiling ncurses for ARM:
********************************
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --with-shared
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR=`pwd`/armbuild

Cross compiling readline for ARM:
*********************************
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR=`pwd`/armbuild

Cross compiling libosip for ARM:
********************************
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR=`pwd`/armbuild

Cross compiling speex for ARM:
********************************
First you need to remove ogg headers from your build system to avoid a dirty
conflict between
your build machine binaries and the arm binaries. They are usually in a
libogg-dev package (rpm or deb).
Then:
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
--enable-fixed-point --enable-arm-asm
make
make install DESTDIR=$ARM_INSTALL_TREE
make install DESTDIR='pwd'/armbuild


Cross compiling linphone for ARM
********************************
First you need to remove all .la files from the ARM_INSTALL_TREE because
it confuses libtool and makes
the linker use your build machine binaries instead of the arm-crosscompiled
ones.
rm -f $ARM_INSTALL_TREE/usr/lib/*.la
#for some reason pkg-config doesn't like cross-compiling...
export PKG_CONFIG=/usr/bin/pkg-config
./configure --prefix=/usr --host=arm-linux --with-gnu-ld --disable-static
\
       --disable-glib --with-osip=$ARM_INSTALL_TREE/usr \
   --with-readline=$ARM_INSTALL_TREE/usr \
       SPEEX_CFLAGS="-I$ARM_INSTALL_TREE/usr/include" \
       SPEEX_LIBS="-L$ARM_INSTALL_TREE/usr/lib -lspeex "
make
```

```
make install DESTDIR='pwd'/armbuild
```

🔔**注意：** 上面编译过程笔者都进行了验证，笔者的安装路径为/usr/local/linphone/linphone_arm，在编译 linphone 之前补充编译下面几项。下面列出的编译配置参数都是非常基本和重要的，如果读者要对 linphone 的其他功能进行安装，则需要读者自己进一步研究。

（1）编译 ffmpeg。

```
# cd ffmpeg-0.5.1
# ./configure --prefix=/usr/local/linphone/linphone_arm --enable-cross-
compile \
--enable-swscale --enable-pthreads --enable-shared --disable-static
# make clean
# make
# make intall
```

（2）编译 oRTP。

编译过程中会遇到将警告作为错误处理，则需要修改对应的 Makefile 中编译参数，修改如下。

```
# vi src/Makefile
CFLAGS = -g -O2 -Wall -Werror  -DORTP_INET6
修改
CFLAGS = -g -O2 -Wall  -DORTP_INET6   # -Werror，该参数是将警告作为错误来处理
# ./configure --prefix=/usr/local/linphone/linphone_arm --host=arm-linux
--with-gnu-ld --disable-static
# make
# make install
```

（3）编译 mediastreamer2。

编译时缺少两个宏的定义，在下面的文件中添加这两个宏的定义。

```
# vi /usr/local/linphone/linphone arm/include/speex/speex preprocess.h
#define SPEEX_PREPROCESS_GET_PSD_SIZE      34
#define SPEEX_PREPROCESS_GET_PSD          35
# ./configure --prefix=/usr/local/linphone/linphone arm --host=arm-linux
--with-gnu-ld \
 PKG_CONFIG_PATH=/usr/local/linphone/linphone_arm/lib/pkgconfig
# make
# make install
```

（4）编译 linphone。

```
# ./configure --prefix=/usr/local/linphone/linphone arm  --host=arm-linux
--with-gnu-ld \
 --disable-static --disable-glib --with-osip=/usr/local/linphone/
 linphone arm \
 --with-readline=/usr/local/linphone/linphone arm \
SPEEX_CFLAGS="-I/usr/local/linphone/linphone_arm/include" \
SPEEX_LIBS="-L/usr/local/linphone/linphone_arm/lib -lspeex" \
--enable-video=no
PKG_CONFIG_PATH=/usr/local/linphone/linphone_arm/lib/pkgconfig
--enable-gtk_ui=no
# make
# make install
```

## 20.9.2　Linphone 的测试

在目录/usr/local/linphone/linphone_arm/bin/下会生成下面的工具。

```
# ls /usr/local/linphone/linphone_arm/bin/
captoinfo  ffmpeg  ffserver  infotocap  linphonecsh  sdl-config  sip reg
speexenc  tic  tput
clear      ffplay  infocmp   linphonec  reset        sipomatic   speexdec
tack      toe  tset
```

移植前首先查看其依赖的库文件，可以通过编译在 X86 目录下的 linphone 进行查看，对应的移植 arm 目录下的 linphone 也需要相应的库文件。

```
# ldd linphonec
       linux-gate.so.1 =>  (0x00898000)
       liblinphone.so.3 => /usr/local/linphone/linphone x86/lib/
       liblinphone.so.3 (0x0034e000)
       libreadline.so.5 => /usr/lib/libreadline.so.5 (0x443c3000)
       libncurses.so.5 => /usr/lib/libncurses.so.5 (0x443f9000)
       libmediastreamer.so.0 => /usr/local/linphone/linphone_x86/lib/
       libmediastreamer.so.0 (0x0053e000)
       libortp.so.8 => /usr/local/linphone/linphone x86/lib/libortp.so.8
       (0x00218000)
       libspeex.so.1 => /usr/local/linphone/linphone_x86/lib/
       libspeex.so.1 (0x0019c000)
       libm.so.6 => /lib/libm.so.6 (0x446c7000)
       libosipparser2.so.4 => /usr/local/linphone/linphone_x86/lib/
       libosipparser2.so.4 (0x003e9000)
       libosip2.so.4 => /usr/local/linphone/linphone_x86/lib/
       libosip2.so.4 (0x00120000)
       libpthread.so.0 => /lib/libpthread.so.0 (0x446f6000)
       libc.so.6 => /lib/libc.so.6 (0x44588000)
       libeXosip2.so.4 => /usr/local/linphone/linphone_x86/lib/
       libeXosip2.so.4 (0x0091e000)
       libdl.so.2 => /lib/libdl.so.2 (0x446f0000)
       libasound.so.2 => /lib/libasound.so.2 (0x44024000)
       libspeexdsp.so.1 => /usr/local/linphone/linphone x86/lib/
       libspeexdsp.so.1 (0x00fe0000)
       libavcodec.so.52 => /usr/local/linphone/linphone_x86/lib/
       libavcodec.so.52 (0x00ff1000)
       libswscale.so.0 => /usr/local/linphone/linphone_x86/lib/
       libswscale.so.0 (0x00132000)
       libSDL-1.2.so.0 => /usr/local/linphone/linphone x86/lib/
       libSDL-1.2.so.0 (0x00232000)
       libX11.so.6 => /usr/lib/libX11.so.6 (0x4481d000)
       librt.so.1 => /lib/librt.so.1 (0x44724000)
       libvorbisenc.so.2 => /usr/lib/libvorbisenc.so.2 (0x43c14000)
       libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0x45143000)
       libgcc s.so.1 => /lib/libgcc s.so.1 (0x45135000)
       libssl.so.6 => /lib/libssl.so.6 (0x43f02000)
       libcrypto.so.6 => /lib/libcrypto.so.6 (0x43d57000)
       /lib/ld-linux.so.2 (0x43bb9000)
       libnsl.so.1 => /lib/libnsl.so.1 (0x00165000)
       libresolv.so.2 => /lib/libresolv.so.2 (0x0017b000)
       libavutil.so.49 => /usr/local/linphone/linphone x86/lib/
       libavutil.so.49 (0x00f8a000)
       libz.so.1 => /usr/lib/libz.so.1 (0x4470f000)
       libXau.so.6 => /usr/lib/libXau.so.6 (0x44818000)
```

```
        libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0x447cf000)
        libvorbis.so.0 => /usr/lib/libvorbis.so.0 (0x43bde000)
        libgssapi_krb5.so.2 => /usr/lib/libgssapi_krb5.so.2 (0x001b2000)
        libkrb5.so.3 => /usr/lib/libkrb5.so.3 (0x002c1000)
        libcom_err.so.2 => /lib/libcom_err.so.2 (0x00110000)
        libk5crypto.so.3 => /usr/lib/libk5crypto.so.3 (0x001dd000)
        libogg.so.0 => /usr/local/linphone/linphone_x86/lib/libogg.so.0
(0x009d2000)
        libkrb5support.so.0 => /usr/lib/libkrb5support.so.0 (0x00113000)
```

移植这些库文件将是一个漫长的过程，但是已经没有什么难题了，缺少的库通过交叉编译后移植到开发板的/lib 目录下或/usr/lib 下，最后的运行结果将和在虚拟机上运行的结果一样。

## 20.10　小　　结

本章主要讲解 VoIP 技术基本概念，讲解其中两个重要协议 SIP 和 RTP，并且对 SIP 协议和 RTP 协议进行源码分析；还通过实例介绍了 X86 和 ARM 平台的编译过程，演示了 X86 平台上的使用方法。其在 ARM 上的使用过程留给读者自己去试验。相信本章对读者的项目有很大的帮助，如果还需要扩展一些功能或者需要其他库文件的支持，需要读者进一步研究。